A Precept-Driven Learning Algorithm

A Thesis

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Christophe G. Giraud-Carrier

April 1993

This thesis by Christophe Giraud-Carrier is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

Tony Martinez, Committee Chair

David Embley, Committee Member

Date

David Embley, Graduate Coordinator

ACKNOWLEDGMENTS

This research was supported in part by a grant from Novell, Inc., and by the Department of Computer Science at Brigham Young University. The computing resources of the Connectionist and Neural Nets Lab were invaluable for the simulations. I wish to express my deepest gratitude to my wife for her constant support and patience. I thank my many friends who have always lent a listening ear to my ideas, in particular Cory Barker. I appreciate the insights of Dr. Tony Martinez and Dr. David Embley who reviewed the early drafts of this thesis. Most of all, I wish to acknowledge the hand of the Lord in this work, as well as in all other aspects of my life. This thesis would not have come to fruition without His help.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Machine learning is an attempt at devising mechanisms that machines can use to learn, rather than being explicitly programmed for, real-world applications. If A is an application having inputs $i_1, \ldots, i_n$ and outputs $o_1, \ldots, o_p$, with domains $I_1, \ldots, I_n, O_1, \ldots, O_p$, respectively, then $I=I_1\times\ldots\times I_n$ is the input space and $O=O_1\times\ldots\times O_p$ is the output space for A. In the context of this research, A is a mapping from I to O, i.e., a function that associates a given set of input values to some unique set of output values. Informally, the values of I can be viewed as objects and the values of O as labels for classes of objects. Because A assigns each object to its proper class, A is a classifier.

A common approach to machine learning, called *training set learning* (TSL) consists of approximating A from a subset of $I\times O$, called the training set. The training set is commonly derived from real-world experimentation. TSL has become very popular for both practical and philosophical reasons. It is practical because, for any real-world application A, it is usually easy to come up with a rich training set for A. It is also philosophically attractive because a training set learner learns on its own in that it is responsible for extracting from the training set the critical features of A without human intervention. These critical features are used in turn to approximate A. Empirical studies have shown that good results can be achieved with TSL [Mooney et al. 90, Kibler and Aha 87, Sejnowski and Rosenberg 87].

However, there are several problems with TSL. Training set learners (e.g., backpropagation) are typically slow as they may require several passes over the training set. Also, there is no guarantee that, given any arbitrary training set, the system will find enough good critical features to get a reasonable (i.e., small misclassification error) approximation of A. Moreover, as the size of the input space increases, TSL becomes more computationally expensive [Blumer et al. 87]. Finally, the scarcity of interesting positive theoretical results suggests that learning without sufficient a priori knowledge is difficult.

This thesis introduces *precept-driven learning* (PDL). PDL is intended to overcome some of TSL's weaknesses. In PDL, the training set is augmented by a small set of *precepts*. A pair $p = (i, o)$ in $I\times O$ is called an *example*. A precept is an example in which some of the i-entries (inputs) are set to the special value *don't-care*. An input whose value is not don't-care is said to be *asserted*. An input i_j whose value is don't-care in some pair p has the following property. Every pair p' obtained from p by asserting i_j to be any one of its possible values (taken from I_j) and leaving the other inputs of p unchanged has the same output as p. Informally, this says that in the context of p, the value of i_j has no effect on the value of the output. The use of the special value don't-care is therefore a short-hand notation that allows a single pair to represent several examples. To be exact, a pair containing don't-care inputs *represents* as many examples as the product of the sizes of the input domains of its don't-care inputs.

A pair p is said to *cover* a pair q if q equals p on all asserted inputs of p. If p covers q, then p represents at least all of the examples that q represents, i.e., there are no examples represented by q that p does not also represent. An example does not cover any other pair (all of its inputs are asserted) and corresponds therefore to a very specific state of the real-world, whereas a precept is more general since it covers several examples.

A precept corresponds to some high-level-knowledge or meta-knowledge one may have about the real-world. It "suggests something advisory and not obligatory communicated typically through teaching" (Webster's Dictionary). Precepts can be thought of as some instantiation of domain knowledge applied to a particular algorithm. They are identical to critical features except that they are given by the teacher and not derived by the algorithm. Moreover, they are not necessarily accurate, but represent some hint or bias. For instance, in an application dealing with how weather conditions affect what one wears, a possible precept would be that if it rains, one takes an umbrella, regardless of whether the sun is also shining, or any other unnecessary (don't-care) detail.

Chapter 2 motivates PDL, discusses related work and shows how the proposed approach addresses issues raised in the context of machine learning and artificial intelligence. Chapter 3 presents the details of a precept-driven learning algorithm (PDLA) capable of handling both correct and incorrect precepts. The approach used follows the judicial system's assertion that an individual is innocent until proven guilty. It consists of always assuming that the precepts are correct until enough evidence may be obtained from the training set to decide otherwise. Chapter 4 describes and analyzes the results of several simulations of PDLA against real-world data sets. In each case, performance is measured in three situations: without precepts (i.e., pure training set learning), with incorrect precepts (i.e., precepts contradicting the actual features of the application), and with correct precepts. Chapter 5 discusses variants of and extensions to PDLA. Finally, Chapter 6 concludes the thesis.

CHAPTER 2

PDL AND RELATED WORKS

In this chapter, we situate PDL in the context of Machine Learning and show how it addresses some of the important issues of this field of research.

Neural Networks, Artificial Intelligence and Machine Learning

Research in Machine Learning is concerned with endowing machines with learning capabilities. Historically, two main areas of research have been devoted to this end, namely, Artificial Intelligence-Machine Learning (AI-ML) and Neural Networks (NNs). AI-ML began in the 1950's (see the introductions to [Dreyfus 79] and Chapter 1 of [Shavlik and Dietterich 90] for good historical reviews). NNs made their first real appearance with Rosenblatt's perceptron in the 1950's (see [Rosenblatt 58, Rosenblatt 62, Minsky and Papert 67]) as a result of earlier studies on neurons [McCulloch and Pitts 43, Hebb 49]. Researchers in AI-ML and NNs had the ambition, hope, and enthusiasm characteristic of most newly-born fields of research. However, the optimism of the early days was soon replaced by the hard reality of the extreme complexity of the task at hand. Both fields met with violent criticism, both of a theoretical nature (e.g., [Minsky and Papert 67]) and of a philosophical nature (e.g., [Dreyfus 79]). The wound was severe, particularly for NNs. However, recent discovery of new learning mechanisms (e.g., multilayer backpropagation [Rumelhart et al. 86]) and availability of better hardware have created a renewed interest in machine learning. In 1980, the First Machine Learning Workshop was held at Carnegie-Mellon University in Pittsburgh, and in 1987, the First International Conference on Neural Networks was organized in San Diego, CA. These events marked the official rebirth of these areas of research. Since 1987, research in these areas has been very active.

The AI-ML approach is symbolic in nature and tends to be driven by the idea of "psychological plausibility," i.e., that the construction of systems should be based on observations of human psychological processes. The NN approach, on the other hand, is connectionist and is driven by the idea of "biological plausibility," i.e., that the construction of systems should reflect the structure of the human brain. Minsky gives a somewhat biased (yet interesting) comparison of these two methodologies in [Minsky 91].

Even though the means are different, AI-ML and NN have the common goal of designing systems capable of learning certain human tasks. It has thus become customary to merge the two disciplines into the area of Machine Learning. In blending the two approaches, the community opened the door to comparative studies [Mooney et al. 89], as well as to attempts at designing systems combining them [Shavlik and Towell 89, Hall and Romaniuk 90, Barker 93, This Thesis]. In the mean time, the recently renewed interest in machine learning has given rise to a new taxonomy of learning mechanisms in general. [Remy 88, Carbonell et al. 83] are good reviews of this taxonomy and [Shavlik and Dietterich 90] contains many of the key papers discussing various facets of machine learning.

PDL and Machine Learning

Machine learning can be divided into several categories, typically including inductive learning [Michalski 83], learning by analogy [Carbonell 83], speedup learning [Mitchell et al. 83] and unsupervised learning [Rumelhart and Zipser 85]. Inductive learning is characterized by the acquisition of knowledge by inductive inferences from examples. Learning by analogy consists of using similarity with known facts or procedures to acquire or reason about new ones. Speedup learning (also known as skill acquisition) explores ways of applying learning to improve the speed with which a problem-solver can solve problems. Unsupervised learning (also called conceptual clustering) is concerned with the automatic clustering of concepts, without any human intervention or a priori knowledge of the concepts.

TSL is a form of inductive learning. Knowledge is acquired (i.e., critical features are discovered) by inferences from the examples which make up the training set. These inferences correspond to the application of various *generalization rules*. A survey of generalization rules and their usage is in [Michalski 83].

PDL is also a form of inductive learning in which the training examples are augmented by precepts. It is an implementation of what has been defined by Michalski as "*incremental concept refinement*, where the input information includes, in addition to the training examples, previously learned hypotheses, or human-provided initial hypotheses that may be partially incorrect or incomplete" [Michalski 83]. Other models implementing incremental concept refinement are scarce. One variant is described in [Michalski and Larson 78].

PDL as a Plausible Approach

PDL appears to be natural in human learning. Seldom do humans learn exclusively from random examples, without ever being exposed to more general rules. Often, these rules correspond to what we regard as *common-sense knowledge*. For example, regardless of most external circumstances, it does not make sense to put your hand on a red-hot stove. This is a general rule or precept that does not require successive examples of various settings involving your hand on the stove, to be acquired. Rather, humans are exposed to precepts taught by others (e.g., "you are going to get burned"), which, together with a built-in pain mechanism, allow the (almost instantaneous) learning of the more general rule. Learning a foreign language's grammar provides another instance in which learning is not limited to confrontation with numerous specific examples. Foreign-language students first learn English rules (e.g., the plural of a noun is formed by tagging an *s* to the end of the singular form of that noun), and are then exposed to the exceptions (e.g., geese instead of gooses, mice instead of mouses).

PDL as a Means to Meet Machine Learning Goals

More important than PDL's plausibility is its applicability to generalization: the current goal of machine learning research. Generalization is the process of presenting the system with only the input part of some *unknown* pair (i.e., not seen during learning) and letting the system guess the output part. An important measure of a system's overall performance is given by how well it generalizes, that is, a measure of its predictive accuracy [Quinlan 86]. If U is a set of unknown inputs and U' is the subset of U for which the system (after learning) predicts the correct output, then the percentage associated with the ratio $|U'|/|U|$, where $|U|$ denotes the number of elements in U, empirically measures the performance of the system.

Generalization is based on the set of critical features available to the system. In TSL, the extraction of critical features is based solely on the training set and is performed by the system alone. Even though several forms of biases exist in TSL (representation, number of hidden nodes, etc.), TSL ignores precepts and essentially denies a human (or another machine) the ability to transfer domain knowledge about the application directly to the system. It seems however that having access to some precepts would speed up learning by pruning the input space. Michalski argues that *problem background knowledge*, of which precepts are a special case, is necessary "to constrain the space of possible inductive assertions" [Michalski 83]. It has also been shown that unbiased generalization programs cannot outperform rote learners, and that "the power of a generalization system follows directly from its biases." [Mitchell 80]. Mitchell identifies several classes of reasonable biases. Precepts are a form of what he calls *factual knowledge of the domain*, except that in PDL factual knowledge is relaxed to include hints or biases that may not be correct. Moreover, the addition of correct precepts obviously increases the probability that the known critical features of the application will be learned. PDL systems can thus be expected to perform better in generalization than TSL systems.

Are There Problems With PDL?

Two concerns arise with PDL: first, PDL requires human supervision and second, there may be other critical features unknown even to humans. The first concern is appeased by realizing that the intent of PDL is not to do a large amount of preprocessing to discover precepts, but rather to only endow the system with some readily available (if any exists) a priori knowledge (or bias) about the application. Moreover, it is hoped that the fast generation of simple precepts can be automated. The second concern is not valid since PDL does not preclude self-learning. The precepts are indeed not sufficient. The training set must still be shown to confirm or invalidate the precepts, as well as to allow the system to potentially discover other critical features.

A Broader Picture

Common-sense knowledge has been argued to be the missing key element in learning systems. The result is what has been referred to as the *brittleness bottleneck* [McCarthy 84, Lenat et al. 90]. Even sophisticated learning systems lack "certain common sense knowledge and ability possessed by any non-feeble-minded human...are difficult to extend beyond the scope originally contemplated by their designers, and...usually don't recognize their own limitations." [McCarthy 84]. Many real-world applications have inherent common-sense rules, and PDL attempts to become less brittle by incorporating them (in the form of precepts) into the learning system directly. Designing general methods that effectively combine prior knowledge with training set data is listed as an open problem in [Mitchell 82]. PDL is one possible (and somewhat specific) answer to this problem.

Why Precepts?

The semantics of the word "precept" carry the idea of some general information being given via teaching. It seems to fit very nicely with the philosophy of this work. The idea is to tell the system things that a teacher (which could be another machine) already knows about the application. This type of knowledge is not very polished, but is a rough estimate of what ought to be expected given the application's context.

The notion of a precept introduced in this work is similar to that of a rule in rule-based (or expert) systems (see for example [Cordier 84]). However, their use is different. Rule-based systems proceed from a fixed set of rules (obtained from an expert) and reason deductively to infer new facts from an initial set of known facts. The underlying assumption is that the rules are correct and that the set of rules is logically consistent (this assumption is sometimes relaxed by using fuzzy logic). Moreover, such systems do not generally learn, although some proposals have been made to enhance them with learning capabilities [Haas and Hendrix 83, Rychener 83].

Several models would allow the presentation of precepts during their learning phase, such as ASOCS [Martinez 86] and ScNets [Hall and Romaniuk 90]. However, early models of ASOCS always give higher priority to incoming knowledge, do not allow logical inconsistency, and do not have any meaningful generalization (except AA1, see [Martinez et al. 93]). Current efforts in ASOCS research are aimed at overcoming some of these limitations. ScNets are networks that learn rules that can later be used by an expert system. They attempt to combine the connectionist and the symbolic approach. The resulting networks are complex and generalization does not appear trivial.

PDL uses precepts to guide the learning phase. The precepts are always assumed to be correct until enough evidence may be gathered from the training set to refute them. Hence, at any time, the overall system contains inherent inconsistencies. As the system learns, new critical features (possibly absent from the precepts) are also discovered. When presented with unknown inputs, PDL generalizes as discussed in Chapter 3.

CHAPTER 3

A PRECEPT-DRIVEN LEARNING ALGORITHM

In this chapter, we describe a precept-driven learning algorithm (PDLA). An earlier version of PDLA (in which precepts were assumed to be correct and possible exceptions were not dealt with) is described in [Giraud-Carrier and Martinez 93] and is summarized in Appendix A.

Before going into the details of PDLA, we first introduce a few preliminary definitions. Input-output *pairs* refer to either examples or precepts, where inputs can either be asserted or don't-care.

<u>Preliminary</u> <u>Definitions</u>

<u>Definitions (relationships between pairs)</u>:

- Two pairs are *equal* if they have the same input values.
- A pair *p* is a *subset* of a pair *q* if *p* and *q* are equal on the asserted inputs of *q*, that is, *p* is a subset of *q* if and only if *q* covers *p*.
- If *p* is a subset of *q*, then *q* is a *superset* of *p*.
- Two pairs are *discriminated* if they differ in the values of at least one asserted input. If they differ exactly on one asserted input, we say that there is a *near_match* between them.
- Pairs that are not equal and do not satisfy the subset, superset or discriminated relations are said to *overlap*.

Note, if pairs are treated as the sets of examples they cover in the input space, then the above subset, superset, equal and overlap relations are the usual ones from set theory.

<u>Definitions (output-dependent relationships)</u>:

- Two pairs are said to be *concordant* if their outputs are equal, *discordant* otherwise.
- A pair *p* contradicts a pair *q* if *p* and *q* are discordant and *p* is subset of, equal to or overlap of *q*.

<u>Definitions (properties of pairs)</u>:

- *size*: the amount of input space covered by the pair. If all the inputs are asserted then size is 1, otherwise, size is the product of the sizes of the domains of inputs whose value is *don't-care* in the pair.
- *num_asserted*: the number of asserted inputs in the pair.

- *specificity*: the ratio *num_asserted* / total number of inputs. Examples have specificity=1 (maximally specific), while precepts have specificity<1 (more general).

A pair can be *generalized* in many different ways (see [Michalski 83] for a list of generalization rules). However, the purpose is always the same. If a pair p covers a set S of examples, then the pair p' resulting from generalizing p (if such a generalization exists) will cover a set S' of examples such that $S \subseteq S'$.

Example 1:

Consider the pairs p1=(<0,0,*,1,*>, <0>), p2=(<0,0,0,1,0>, <1>), p3=(<0,*,*,1,1>, <1>), p4=(0,0,1,1,1>, <0>), p5=(<0,0,1,1,0>, <1>), where all variables are Boolean.

Then, the following illustrates the above definitions, relative to these pairs:

- p1 is a superset of p2, p4 is a subset of p3, p1 and p3 overlap, p2 and p4 are discriminated, there is a near_match between p2 and p3, there is a near_match between p4 and p5.
- p2, p3 and p5 are concordant, p4 and p5 are discordant.
- p2 contradicts p1, p4 and p5 do not contradict each other since they are discriminated.
- size(p1)=size(p3)=4, size(p2)=size(p4)=size(p5)=1.
- num_asserted(p2)=num_asserted(p4)=num_asserted(p5)=5, num_asserted(p1)=num_asserted(p3)=3.
- specificity(p2)=specificity(p4)=specificity(p5)=1, specificity(p1)=specificity(p3)=.6.

Definitions (attributes of nodes):

PDLA implements a network of simple nodes. Each node stores a pair p and a list of all pairs that contradict p.

Nodes have the following attributes:

- *num_matched*: the number of asserted inputs of the stored pair that match the current network inputs.
- *activation*: the node's activation defined as *num_matched* / *num_asserted*.
- *num_exceptions*: the cumulative sizes of overlap with pairs that contradict the stored pair.

The network architecture used by PDLA is a simple binary tree architecture. The only requirement is for a broadcast and gather facility. Both operations are O(log$n$) where n is the number of nodes in the tree.

The tree starts empty and dynamically adapts to new knowledge. During learning, nodes are added and deleted. When a node is deleted, transformations are applied to the tree to keep it balanced. When a node is added, its position in the tree is determined so as to maximize breadth. Therefore, the tree is always maintained as full as possible.

Example 2:

The pairs are those of Example 1. Suppose the current network consists of the nodes n1, n2 and n3 containing p1, p2 and p3 respectively. The network is shown in Figure 1:

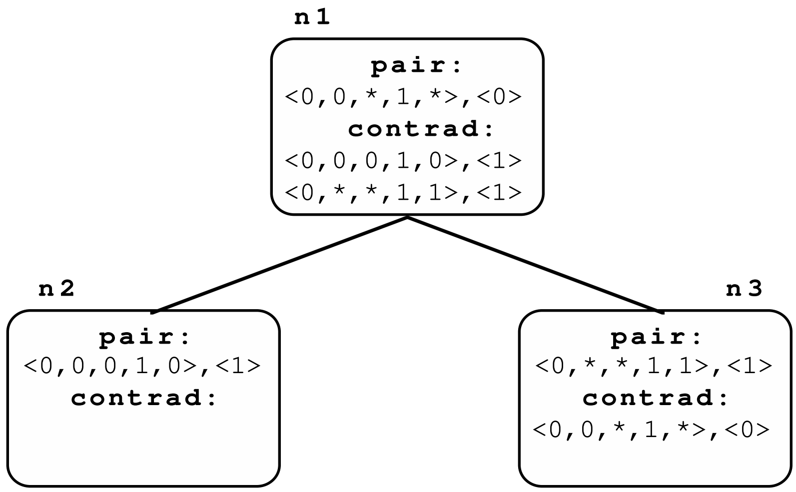

Figure 1 - A Sample Network

We have num_exceptions(n2)=0, num_exceptions(n3)=2, and num_exceptions(n1)=3. Suppose the input <0,0,1,1,0> is presented to the network of Figure 1. Then num_matched(n1)=3 and activation(n1)=1, num_matched(n2)=4 and activation(n2)=.8, num_matched(n3)=2 and activation(n3)=.67.

PDLA Overview

PDLA can run in two modes: the learning mode and the execution mode. During learning, training pairs are presented to the network. For each new training pair, a node is tentatively created, and the pair is broadcast to all the other nodes. Each node reacts in some way to the new pair. It may:

- self-delete,
- update its num_exceptions counter,
- generalize,
- cause the new node to be deleted, or
- cause the new pair to be generalized.

The decision of which action to be performed by each node is determined by the relation of that node's stored pair to the training pair. A new node is always created unless the network contains a node whose stored pair is concordant with the incoming pair and either covers the incoming pair or generalizes with the incoming pair. All existing nodes that can generalize do so. In the case where the training pair must generalize, if more than one node cause it to generalize, only one possible generalization is chosen at random, and the node responsible for it self-deletes.

During execution, the network's inputs are set and each node computes its activation which is a measure of how closely its stored pair matches the network's inputs. The node with highest activation wins the competition and the output part of its stored pair becomes the output of the network. In case of a tie, priority is first given to the most specific pair (one nearest-neighbor) and then to the one that has the least amount of exceptions.

The following sections describe these modes in detail and discuss, via an example, some of the issues raised by PDLA and how they were addressed.

The Learning Algorithm

We first present the algorithm and immediately show an example of its execution. When learning a new application, the network starts empty. Training pairs are presented to the network in an incremental fashion, and learning proceeds as follows.

The *create_flag* keeps track of whether a new node must be created for the current training pair and the *modified_flag* keeps track of whether the current training pair is a candidate for modification by generalization.

```
Begin  PDLA
For each training pair q
  create a new node nn for q with num_exceptions set to 0
  set create_flag to true
  set modified_flag to false
  For all nodes n (storing pair p) in the current network
    case superset(n.p, q)
      if concordant(n.p, q) then set create_flag = false
      else Update_Num_Exceptions(n)
      endif
    case equal(n.p, q)
      if concordant(n.p, q) then set create_flag = false
      else n self-deletes
      endif
    case subset(n.p, q)
      if concordant(n.p, q) then n self-deletes
      else Update_Num_Exceptions(nn)
      endif
    case overlap(n.p, q)
      if discordant(n.p, q) then
        Update_Num_Exceptions(nn)
        Update_Num_Exceptions(n)
      endif
    case discriminated(n.p, q)
      diff = abs(num_asserted(n.p) - num_asserted(q))
      if concordant(n.p, q) & near_match(n.p, q) & diff<=1
      then
        if size(n.p)>=size(q) & num_asserted(n.p)>1 then
          drop variable in n.p
          set create_flag to false
        else if num_asserted(q)>1 then
          drop variable in q to get q'
          set modified_flag to true
        endif
  EndForAll /* all nodes */
  if create_flag then
```

```
    if modified_flag then
      randomly select one of the nodes which set the flag
      replace q by q' in nn
      delete the selected node
    endif
    retain nn
  else delete nn
  endif
EndForEach /* each training pair */
Process_Exceptions()  /* see details below */
```

End PDLA

Note that the algorithm automatically handles precepts. All possible relations between all existing nodes and the current training pair are checked. Modifications caused by dropping of variables and updating of exceptions are made both to the existing nodes and the training pair. It follows that precepts can be entered in at any time during the learning phase. However, since PDLA is incremental, the order of presentation during learning may have an impact on the outcome. Showing precepts first may improve the speed of learning.

Note also that parallelism is inherent in the algorithm. All the nodes can execute the "For all nodes..." loop in parallel. With the tree architecture discussed above, this allows reduction of the complexity of PDLA as shown below.

Detailed Example of PDLA

Consider an application with 5 Boolean inputs and a single Boolean output. Let the training set consist of the following pairs:

- p1 = (<0,0,*,1,*>,<0>),
- p2 = (<0,0,0,1,0>,<1>),
- p3 = (<0,0,1,1,1>,<0>),
- p4 = (<1,0,0,1,1>,<1>),
- p5 = (<1,0,0,1,1>,<0>),
- p6 = (<1,1,0,1,1>,<0>),
- p7 = (<0,*,*,1,1>,<1>).

We assume that these pairs are presented to the system in the order p1 through p7. Note that p1 and p7 are both precepts. For each pair in the training set, we show what PDLA does and discuss why. The network starts empty.

1. Presentation of p1

A new node is created for p1 and since the network is empty, the create_flag and modified_flag remain as initialized. It follows that the new node, call it n1, is retained. The resulting network is in Figure 2.

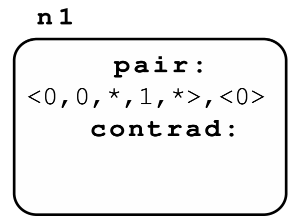

Figure 2 - Network After Presentation of p1

2. <u>Presentation of p2</u>

A new node is created for p2. Node n1 executes the "For all" loop. Its stored pair is found to be superset and discordant. The function Update_Num_Exceptions is applied to n1, and the new node, call it n2, is retained. The resulting network is in Figure 3.

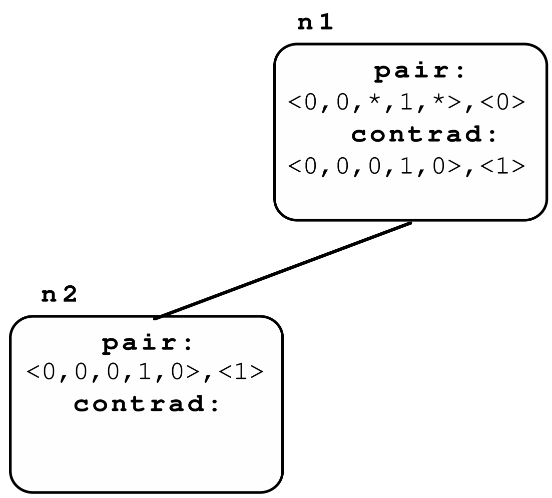

Figure 3 - Network After Presentation of p2

To be able to invalidate precepts and more generally to deal with exceptions, PDLA must keep track of these exceptions (function Update_Num_Exceptions above). To that end, each node keeps a list of all the pairs that contradict (i.e., are exceptions to) its stored pair. However, there is an important difficulty to be addressed, namely that of redundancy. If a pair p that has been contradicted by a pair q is later contradicted by a pair q' that is either a subset of, superset of or equal to q, then certain precautions must be taken.

If q' is equal to q, PDLA discards q', that is, identical exceptions count only once. If q' is a subset of q, PDLA discards q' since q covers q' and therefore already accounts for the exceptions due to q'. If q' is a superset of q, PDLA replaces q by q' (this is the dual of the previous case). Since no discriminated pair can give rise to exceptions, the list at each node contains only pairs that are related by overlap. Hence, num_exceptions is easily obtained by adding together the amount of overlap of input space between each pair in the list and the stored pair. The function Process_Exceptions decides what must be done with exceptions and is discussed below.

PDLA only accounts for exceptions that are actually seen in the training set. This guarantees that general rules involving multistate inputs are not invalidated too hastily due to our strong generalization mechanism (see Paragraph 5 below).

Note that keeping a list at each node is expensive in terms of storage, and that the process of updating the list is also computationally expensive (linear in the size of that list). However, it is extremely difficult to circumvent this problem, which is felt must be addressed for PDLA to be of any use. Moreover, unless precepts are really bad (i.e., inconsistent with the training set), the list of exception pairs is expected to be small.

3. <u>Presentation of p3</u>

A new node is created for p3. Node n1 and n2 execute the "For all" loop in parallel. The pair in node n1 is found to be superset and concordant, so the create_flag is set to false. The pair in n2 is found to be discriminated and discordant, so no changes are caused by n2. It follows that the node created for p3 is deleted and the network after p3 is identical to the network of Figure 3.

Note that in this case, it is consistent with intuition not to create a new node for a pair that is a subset of and concordant with another pair already present in the network.

4. <u>Presentation of p4</u>

A new node is created for p4. Node n1 and n2 execute the "For all" loop in parallel. The pair in node n1 is found to be discriminated and discordant, so no changes are caused by n1. The pair in n2 is found to be discriminated and concordant, but there is no near_match between the pairs, so no changes are caused by n2. It follows that the node created for p4, call it n3, is retained. The resulting network is in Figure 4.

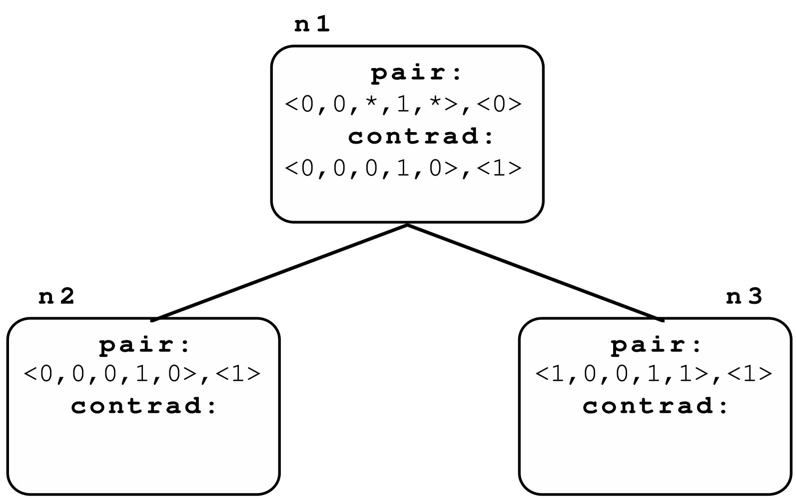

Figure 4 - Network After Presentation of p4

If there does not seem to be a natural way for the current network to predict the output of the incoming training pair, then a new node must be created for that pair. In PDLA, natural means that either there exists a node whose pair is concordant superset, or there exists a node that causes the incoming pair to generalize. Neither is true for p4, so n3 must be added.

5. Presentation of p5

A new node is created for p5. Nodes n1, n2 and n3 execute the "For all" loop in parallel. The pair in node n1 is found to be discriminated, concordant and near_match, but diff=2>1, so no changes are cause by n1. The pair in n2 is found to be discriminated and discordant, so no changes are caused by n2. The pair in n3 is found to be equal and discordant so n3 will self delete. It follows that the node created for p5 is retained. We call it n3 since the current n3 self deletes. The resulting network is in Figure 5.

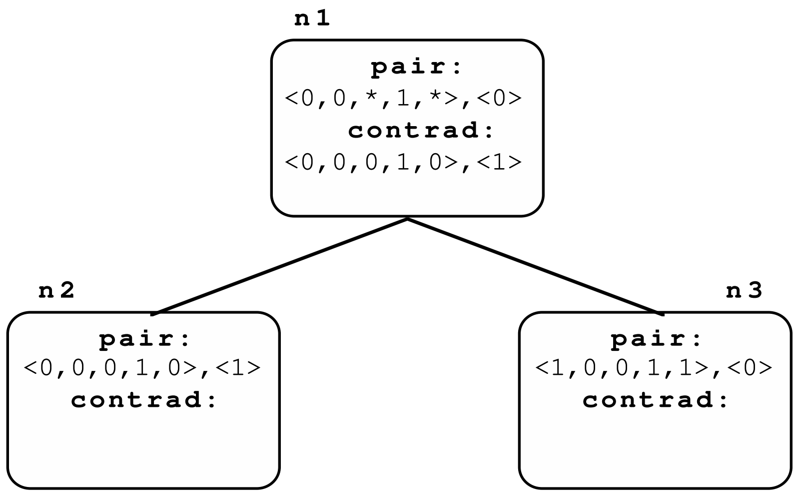

Figure 5 - Network After Presentation of p5

This example shows that PDLA does not handle noise. In this work, *noise* is defined as discordant equal pairs. Another case of noise, which this thesis does not address, is encountered when the application to be learned is a distribution rather than a function. PDLA does not handle noise in any meaningful way. If two pairs are found to be noisy, only the later one is retained. In Chapter 5, we discuss one possible way to extend the current model to better handle noise.

6. Presentation of p6

A new node is created for p6. Nodes n1, n2 and n3 execute the "For all" loop in parallel. The pair in node n1 is found to be discriminated, concordant, but there is no near_match between the pairs, so no changes are caused by n1. The pair in n2 is found to be discriminated and discordant, so no changes are caused by n2. The pair in n3 is found to be discriminated, concordant, near_match and diff=0≤1 so generalization will take place. Since the first test is satisfied, the unique variable on which the two pairs differ is dropped in the pair stored in n3 and the create_flag is set to false. It follows that the node created for p6 is deleted, but n3 is modified. The resulting network is in Figure 6.

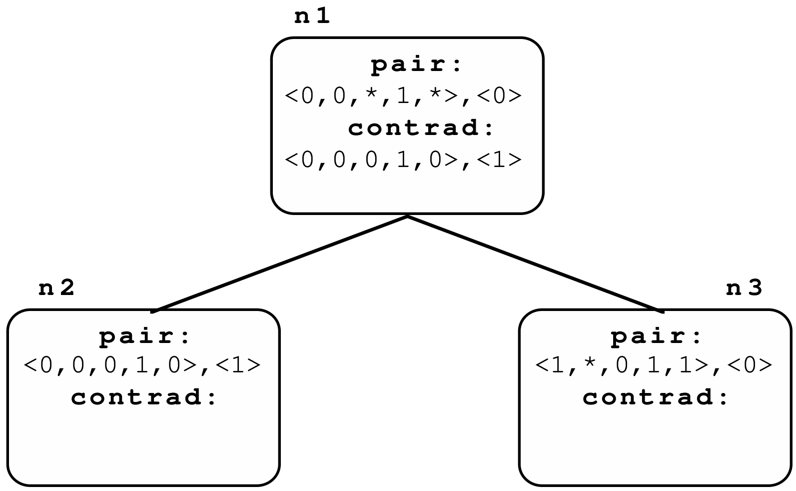

Figure 6 - Network After Presentation of p6

PDLA implements only the dropping condition generalization rule (see [Dietterich and Michalski 83]). In the context of our work, this rule is applied whenever two pairs are concordant, differ in the value of exactly one asserted input out of all inputs asserted in both pairs (function near_match above), and one has zero or one more asserted input than the other (variable diff above). The rule is applied by replacing the differing input by the special value don't-care in the pair that has the largest size, as long as num_asserted is greater than 1. The condition on relative sizes guarantees that generalization is maximal. The condition on num_asserted guarantees that no rule will be created whose inputs are all don't-care and that therefore covers all possible examples in the input space (the output of most applications is a non-trivial function of its inputs).

If the input being dropped is Boolean and diff=0, then applying the above general rule corresponds to the logical simplification: (*a* and *b*) or (*a* and not *b*) implies *a*. If the input is multistate, then the rule generalizes from 2 values to all of them, which may be very strong in some instances. In this case however, possible exceptions would be handled naturally by PDLA (see discussion above).

In the earlier version of PDLA (see Appendix A), generalization was performed on pairs having precisely the same number of asserted inputs (i.e., diff=0). This condition has been relaxed here to allow a difference of 1 in the hope of generating more general pairs. We discuss this choice in the Boolean case.

Let *p* and *q* be two concordant, near_match pairs. As seen above, with diff=0, generalization is equivalent to Boolean simplification, so that the new rule is only a short-hand to represent both *p* and *q* (e.g., (<1,0>,<0>) and (<1,1>,<0>) result in (<1,*>,<0>)). When diff=1, since the variable is always dropped in the largest pair, generalization results in a pair that covers both *p* and *q* as well as at least one other pair that has not yet been seen by the system. For example, (<1,*,0>,<1>) and (<1,0,1>,<1>) result in (<1,*,*>,<1>) which also covers the pair having input <1,1,1>. In general, for diff=*n*, generalization results in a pair that covers both *p* and *q* as well as at least 2^n-1 unseen pairs. A value of diff=0 does not result in any generalization, only a smaller network. A value of diff greater than 1 tends to give very strong generalization (this gets even worse as we consider variables with more than two values). One would argue however that incorrect generalization would be dealt with by PDLA via exception handling. Though this is true, processing exceptions is an expensive operation as seen above. The value of 1 was empirically found to give the best results.

PDLA allows all the existing pairs that can to generalize, thus maximizing overall generalization. However, if the training pair must generalize, it allows it to only drop a single variable. This decision facilitates implementation.

7. <u>Presentation of p7</u>

A new node is created for p7. Nodes n1, n2 and n3 execute the "For all" loop in parallel. The pair in node n1 is found to be overlap and discordant, so num_exceptions is updated for both n1 and the node created for p7. The pair in n2 is found to be discriminated, concordant and near_match, but diff=2>1, so no changes are caused by n2. The pair in n3 is found to be discriminated, discordant, so no changes are caused by n3. It follows that the node created for p7, call it n4, is retained. The resulting network is in Figure 7.

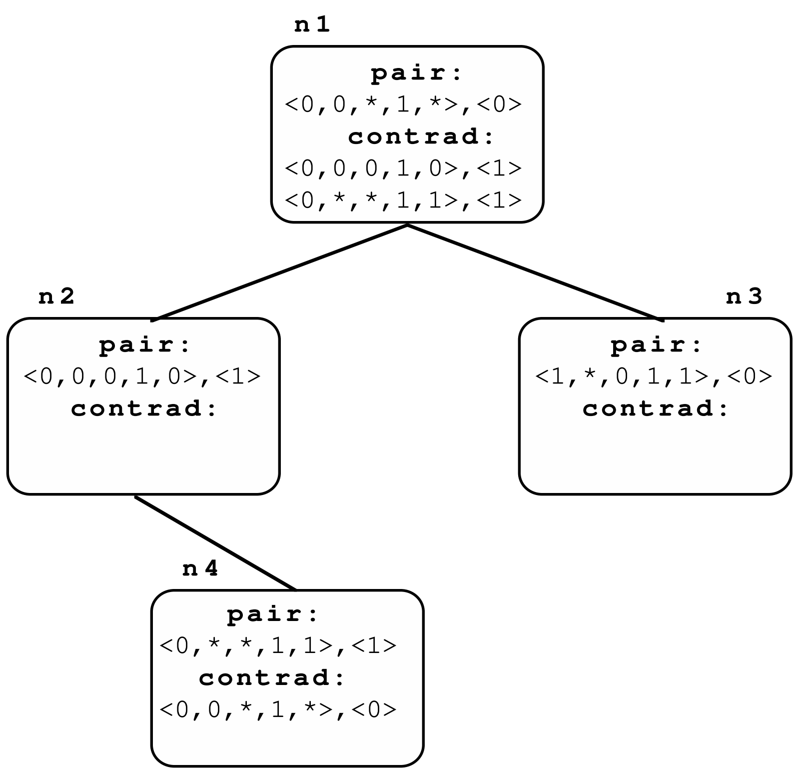

Figure 7 - Network After Presentation of p7

8. <u>Process Exceptions()</u>

When all pairs have been presented, PDLA processes exceptions. The idea is as follows. Precepts are always assumed to be correct until enough evidence is available to refute them. This idea of taking as facts only what is provided, and only modifying knowledge as necessary is a form of what McCarthy calls *circumscription* [McCarthy 77, McCarthy 80, McCarthy 86]. In PDLA, enough evidence is defined as follows (function Process_Exceptions() above). Any node whose num_exceptions exceeds half of the amount of input space covered by its stored pair (i.e., the size of the pair) is removed from the network as being incorrect. The threshold of half seems reasonable, and is confirmed by empirical results. A smaller threshold would result in the deletion of general rules in favor of their exceptions, while a larger threshold might not delete enough incorrect rules.

Using the computation described in Paragraph 2, node 1 whose stored pair has size 4 is such that num_exceptions(n1)=3>2=4/2, so that n1 is deleted. Node n4, on the other hand is such that num_exceptions(n4)=2≤4/2, so that n4 is not deleted. The final network is given in Figure 8.

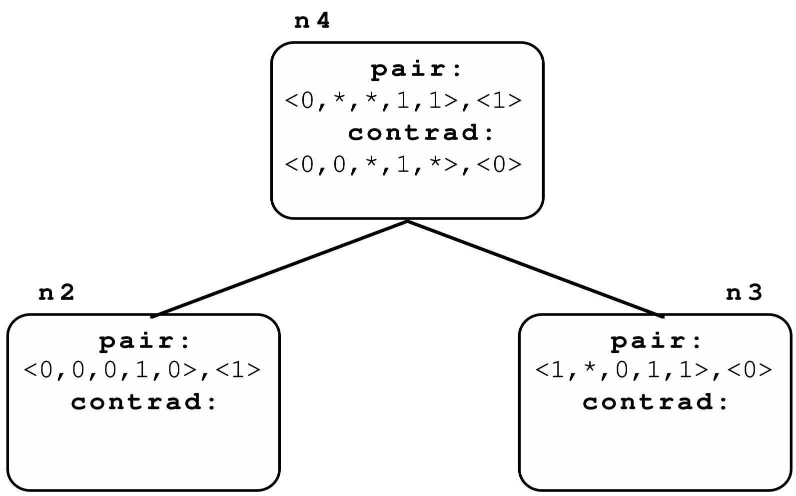

Figure 8 - Final Network

<u>The</u> <u>Execution</u> <u>Mode</u>

In execution mode, the network is presented some unknown inputs and must guess the correct output. When a new pair is presented to the network, each node computes its activation, and a winner is determined as follows:

```
Begin  Exec
  if there is a single node with highest activation, it wins
  else if two nodes n and m compete (same activation), then:
    if specificity(n.p) > specificity(m.p) then n wins
    else if specificity(n.p) = specificity(m.p) then
      if num_exceptions(n) < num_exceptions(m) then n wins
      else m wins
    else m wins
```

  The output of the network is set to the output of the pair stored in the winning node

End Exec

Note that execution can be made very efficient with the tree topology. Implementation and complexity are discussed in more detail below.

<u>Detailed Example of Exec</u>

We discuss how the execution takes place via the following example. Consider the network of Figure 9.

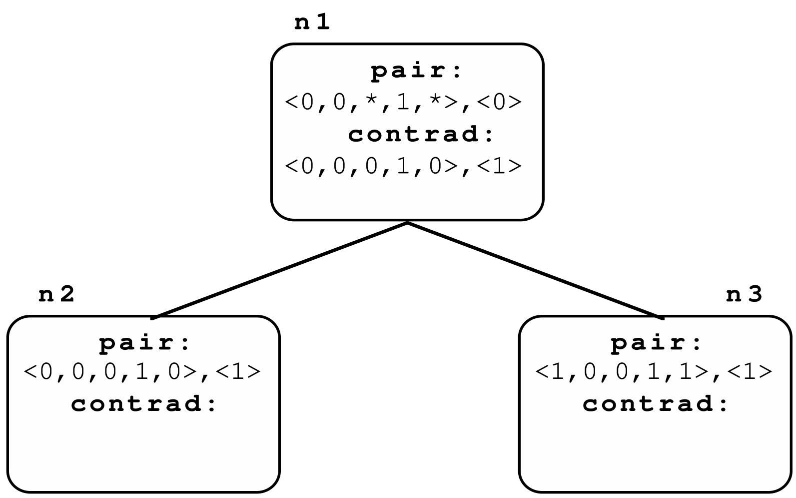

Figure 9 - Example Network for Execution

1. <u><0,0,0,1,1> is Presented</u>

All three nodes n1 through n3 compute their activation. The results are 1, .8, and .8, respectively. In this case, n1 wins and the network output is 0. This result appears reasonable given that we have a precept covering the new input and the new input is not an exception to it.

2. <u><0,0,0,1,0> is Presented</u>

The activations are 1, 1, and .6, respectively. Nodes n1 and n2 are tied. Since n2 is more specific, it wins and the network output is 1, which is what we would expect since the new input is one of the known exceptions to the precept in n1.

When two nodes are tied for highest activation, priority is given to the one whose stored pair is more specific. This is consistent with the philosophy of PDLA. If a new pair is covered by a very general rule, but yet is equal to a more specific pair, then the new pair is one of the exceptions to the more general rule and should therefore be recognized as such. If the new pair is covered but not equal to one of the seen exceptions, then its output will be determined by the general rule. PDLA thus attempts to implement the intuitive idea that "if I have not been told that you are an exception, then you must follow the general rule; but if you are an exception, I will treat you as such." Empirical results show that giving priority to the more specific pair gives better performance than giving priority to the more general pair.

The advantage of this priority scheme is its ability to deal with exceptions. Its disadvantage is that it learns noise too. This is well seen in the results from running PDLA against the Monk3 problem (see Chapter 4).

Priority to Less Exceptions

When two nodes are tied and have the same degree of specificity, priority is given to the one which has seen the least amount of exceptions. This allows the new input to be generalized to the output that is most likely to be correct, and is a major improvement over the random choice of the earlier version (see Appendix A).

Implementation and Complexity

Any efficient architecture that supports broadcast and gather mechanisms can be used. It is suggested that a binary tree architecture be used. The system would then perform as follows.

In learning mode, the training pair is broadcast down the tree to every node. Each node, in parallel, executes the algorithm and returns to its parent node the create_flag, the modified_flag and the possibly modified training pair. As this information is brought back to the root, every node does the following.

- If the node or one of its children returns the create_flag set to false, then the create_flag is passed on and the other information can be discarded.
- If neither the node nor its children nodes return the create_flag set to false, then the modified_flag is tested.

    a. If exactly one of the node or its children returns the modified_flag set to true, then that node is marked and the modified_flag is passed on together with the corresponding modified training pair.

    b. If the modified_flag has been set to true by more than one of the nodes (self and children), then one of them is arbitrarily chosen as the winner. That node is marked and the modified_flag is passed on together with the corresponding modified pair. The nodes that did not win are unmarked if necessary.

At the root of the tree, the upcoming information is used to decide what must be done:

- If the create_flag is false, then the next training pair may be broadcast.
- If the create_flag is true however, then a new node must be created.

    a. If the modified_flag is false then the training pair is stored in the new node.

    b. If the modified_flag is true then the upcoming modified pair is used instead of the original training pair. A token is also sent back down, along the unique path of marked nodes starting at the root, to the lowest-level marked node to cause it to self-delete.

To process the exceptions, a special token is sent to all the nodes and all those who satisfy the condition for self-deletion (see above) actually self-delete.

Computational complexity at the node is at most linear in the number of variables (both input and output), when no update is necessary to the list of exceptions. Complexity of Update_Num_Exceptions is O($v*e$) where v is the number of variables and e is the number of exceptions in the list. The problem with the current implementation is that in theory, e can be exponential. We do not however expect it to be the case in practice, since precepts are expected to be good hints, rather than random guesses.

Ignoring the time necessary to compute at the node (either processing the exception list or various updating), as well as the problems of tree maintenance (which are beyond the scope of this thesis), the learning time is O($p\log n$) where p is the number of pairs presented during learning (i.e., sum of the number of precepts and the size of the training set) and n is the number of nodes in the network. Note that n is bounded by p. The storage requirement at each node consists of one pair, a list of exception pairs, and a few bits for communication. Overall, the memory requirement is thus O($n*e*v$).

In execution mode, the unknown input values are broadcast down the tree to every node. Each node, in parallel, computes its activation. The output of the network is brought back to the root node as follows. Each node sends its activation to its parent node. The parent node compares its own activation and its children's activations and passes on the largest one, together with the value of the output specified by the winner. In case of conflict, the parent node first gives priority to the node containing the most specific pair, and then to the node that has seen the least amount of exceptions. The output value reaching the root node is the output value of the network for the unknown input.

Again, ignoring the costs of local computations (which is O(v) at each node), the execution phase is readily seen to be O($\log n$).

CHAPTER 4

SIMULATION RESULTS

In this chapter, we present and discuss the results of simulations of PDLA against real-world data. The details of the implementation used in this thesis are found in Appendix B.

PDLA was tested on several real-world applications, drawn from the Irvine machine learning database [Murphy and Aha 92]. Two modifications were made however. All real-valued attributes were mapped into multistate (discrete) attributes before simulation (using a K-means algorithm on each individual input), and all don't-know values are treated as don't-care values.

Note that lenses is difficult to learn because of the small size of the data set, and hepatitis is also typically considered hard to learn due to missing data, etc. Performance results were collected in two situations for each application: with and without precepts. The number and generality of the precepts used were varied and are shown below. Each of the percentages given here is an average over 10 runs of PDLA.

In each run, the training set and test set are randomly regenerated, while the set of precepts remains unchanged. In all cases, precepts are presented first, followed by the training set.

Both the training set and the test set contain one half of the examples in the complete data set. This setting is consistent with Mooney et al.'s experiments, except that they used two-third of the data set as training examples and the remaining one-third as a test set [Mooney et al. 89].

The precepts were obtained either from the author's common sense knowledge about each application's domain (Lenses, Zoo, Hepatitis) or from a priori knowledge about the problems (Monk3, Iris, Voting). For example, in the lenses application, it makes sense that someone with a reduced tear production rate should probably not be fitted contact lenses.

Each application was tested with correct and incorrect precepts. The incorrect precepts were simply obtained from the correct ones by changing the value of their outputs. The correct precepts are as follows:

- Lenses (1 precept): if tear production rate is reduced then the patient should not be fitted with contact lenses.
- Zoo (1 precept): if an animal has feathers then the animal is in class 2.
- Voting (1 precept): if a representative voted "no" on the issue of "physician-fee-freeze" then the representative is a democrat.

• Iris (6 precepts): if petal length is 0 (between 0 and 1) then the iris is an Iris Setosa; if petal width is 0 then the iris is an Iris Setosa; if petal length is 5 (between 5 and 6) then the iris is an Iris Versicolour; if petal width is 5 then the iris is an Iris Versicolour; if petal length is 8 (between 8 and 9) then the iris is an Iris Virginica; if petal width is 8 then the iris is an Iris Virginica.

• Hepatitis (2 precepts): if an individual is 21 to 30 years old then the individual lived; if an individual is 51 to 60, is a male, has used steroids, has malaise, has a big and firm liver, has high bilirubin (4.00) and high albumin (6.0) then the individual died

• Monk3 (2 precepts): if jacket_color is blue then the robot is not in the class; if body_shape is not octagon then the robot is not in the class.

The simulation results are summarized in the next two sections. The two numbers following the percentage (%) of correct guesses (i.e., predictive accuracy) correspond respectively to the size of the training set (TS), including precepts where applicable, and the size (i.e., number of nodes) of the resulting network (NS) after completion of the learning phase.

PDLA and Good Precepts

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	67.5	12	6	78.3	13	4
zoo	95.8	45	22	95.8	46	19
voting	88.4	217	73	90.9	218	31
iris	92	75	33	92.7	81	22
monk3	82.3	216	76	85.6	218	43
hepatitis	77.1	77	71	79.5	79	61
mushroom	100	4062	1499	(unnecessary)		

Table 1 - PDLA-Good Precepts

These results show that the addition of simple precepts improves generalization performance and in most cases, significantly reduces the size of the network (greater parsimony). Note that these results also represent an improvement over those of PDLA's earlier version reported in Appendix A.

PDLA and Bad Precepts

To validate our approach to exception handling, PDLA was tested with the bad precepts in two settings. In the first, exceptions are not dealt with (i.e., the function Process_Exceptions() in the PDLA algorithm is skipped altogether). In the second, exceptions are handled as discussed in Chapter 3. Results are shown below in that order.

	No precepts	With precepts

Application	%	TS	NS	%	TS	NS
lenses	67.5	12	6	37.5	13	7
zoo	95.8	45	22	85.1	46	23
voting	86.9	217	75	62	218	74
iris	92	75	33	68	81	38
monk3	82.3	216	76	73.6	218	76
hepatitis	77.1	77	71	61.4	79	72

Table 2 - PDLA-Bad Precepts/No Exception Handling

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	67.5	12	6	50.8	13	6
zoo	95.8	45	22	85.1	46	23
voting	88.4	217	74	81.8	218	71
iris	92	75	33	68	81	38
monk3	82.3	216	76	81.8	218	74
hepatitis	77.1	77	71	61.4	79	72
mushroom	100	4062	1499	(unnecessary)		

Table 3 - PDLA-Bad Precepts/Exception Handling

These results show that PDLA has the ability to deal with bad precepts. In all cases, generalization is better (or the same) when exceptions are processed than when they are not. In most cases, the "recovery" is good, but is obtained at the expense of a larger network. The results on lenses is somewhat significant because the bad precept covers exactly half of the input space, and even with a very small training set, PDLA was able to achieve a little bit better than 50% in generalization.

CHAPTER 5

VARIANTS AND EXTENSIONS

In this chapter, we discuss possible variants of and extensions to PDLA. Some of the variants have been simulated and corresponding results are found in Appendix C.

Variants of PDLA

There are several ways to modify PDLA. The first one that comes to mind is to give priority to more general rules than specific ones when resolving activation conflicts. This, of course, goes against the basic philosophy of PDLA. Results with this approach are similar to results with PDLA in terms of network size, but are almost always worse in terms of generalization. The notable exception is, of course, Monk3, which contains a fair amount of noise that, as was discussed in Chapter 3, PDLA does not handle very well (since it tends to learn it).

Another variant consists of making PDLA more inductive by allowing the current value of the network for the training pair to be used to first set the create_flag. This should tend to bias the learning towards more parsimonious networks. The simulation results confirm this expectation. However, this greater parsimony is gained at the expense of some accuracy in generalization. It is interesting to note, though, that often the loss of accuracy is small compared to the gain of parsimony. For example, the voting application with no precepts goes from 74 nodes to only 19, while generalization goes only from 88.4% to 87.3%. Depending on the goals of the system, this may be a viable option.

It is possible to push further in the direction of inductive learning by modifying PDLA as follows. Preface PDLA by an execution of the current network on the new training pair's input. If the network's output is the same as the training pair's output, then only do generalization and replacement of more specific concordant nodes by the training pair (a form of generalization at this level). If the network's output is incorrect, then proceed with PDLA. Again, as in the previous, greater parsimony is expected. Results show that the gain in parsimony is similar to that of the above approach, while generalization is somewhat worse. This can be explained by the fact that the learning mechanisms of PDLA are not directly related to the way the system executes. In other words, PDLA's modifications in learning mode are not driven by the need to respond to the current system's output. In the case of the Mushroom application, parsimony is reduced by two orders of magnitude (from 1499 to 27 nodes), while generalization is hardly affected (from 100% to 99.9%). Hence, it appears that this approach would be best suited for very consistent, simple applications. A major advantage of this technique is also in the speedup it offers if correct precepts are shown first. In this case, PDLA will run less often and when it does run, it will run faster since the size of the network will always be small.

Extensions to PDLA

It is possible to extend PDLA so that newly generated general rules are rebroadcast to the network. This approach may improve generalization but at the cost of greater algorithmic complexity. Moreover, a decision would have to be made as to when to stop rebroadcasting. It could either go on until no more pairs are generalized or a threshold value could be arbitrarily set on the number of times broadcast is allowed. This extension was not implemented.

It is also possible to extend PDLA so that noise is better handled. For example, the system could keep track of the number of pairs whose inputs are identical as a function of their output, and retain for generalization only the output of the most numerous group.

Other extensions would be necessary to make sure that no information is lost when the function Process_Exceptions causes nodes to be deleted. For example, instead of simply deleting a "bad" precept (see Chapter 3), PDLA could do one of the following. It could keep the bad precept but change its output value, thus creating a new possibly correct precept, or it could replace the bad precept by one or more precepts having the same output, but that cover that subspace of input space on which the bad precept has not been contradicted, thus avoiding the deletion of information that may still be useful. Though somewhat intuitive, these approaches would need to be submitted to the kinds of simulations described in Chapter 4 for a better evaluation. The problem of exception handling is certainly not trivial.

A more effective way of handling exceptions at each node (other than a list) would also increase speed of execution and reduce storage requirements. What is needed is an efficient *representation* of the cover relation, that allows each node to "see" what subspace it covers and which portions of that subspace it is contradicted on.

CHAPTER 6

CONCLUSION

In this concluding chapter, we summarize the results of this thesis and discuss further research issues related to those raised by this work.

Summary

This thesis introduced the notion of precept-driven learning. This new approach, aimed at enhancing the training set approach, was placed in the overall context of machine learning and arguments were given to support its plausibility and adequacy in achieving the goals of this field of research. Based on these arguments, a precept-driven learning algorithm (PDLA) was introduced and discussed in detail. Results of several simulations on real-world data sets were reported.

The main characteristics of PDLA include:

1. Neurally inspired architecture that allows the use of massive parallelism and self adaptation.
2. Bounded learning and execution time complexity.
3. Use of precepts resulting in faster learning, increase in parsimony, and improved generalization.
4. Ability to deal with incorrect precepts.
5. No meaningful handling of noise.
6. Potentially large memory requirement at each node.

Simulation results show that when given correct precepts, predictive accuracy is indeed improved and greater parsimony is achieved in the resulting network. They also show that the system can recover from bad precepts (i.e., refute them) by looking at the training set.

The version of PDLA described in this thesis is not claimed to be the best in any way. It is only intended to be an initial case study, to test some of the intuition discussed in the first two chapters. Several extensions are suggested in Chapter 5. Together with others, they would lead to an overall improvement of PDLA.

Future Work

In this work, precepts have been assumed to be somehow available to the system. In the simulations, the author's common sense was used as the main actor in obtaining precepts. Even though this is a reasonable approach at first, one would surely hope that the generation of precepts could be automated. Notice that precepts are obtained from knowledge about the features of the application rather than from the values of these features as is the case in data-driven approaches. In other words, in order to generate precepts, a machine should have available a vast amount of common sense knowledge about the world in general, the specific context of each application, and the contextual relationships among the applications' features. For example, in the hepatitis application, in order for a machine to determine that the size and aspect of the liver may have an impact on the outcome, it must first know that hepatitis is a disease that affects the liver, and also that a normal (i.e., healthy) liver is typically neither big nor firm.

The issues of common sense and its necessary use if we are to build machines that perform human tasks effectively have long been raised and hotly debated. Attempts have been made at compiling large databases of common sense facts [Lenat et al. 90] and formalizing common sense [Lifschitz 91, Hobbs and Moore 85, Davis 90]. Most formalizations are variants of first-order predicate logic. These schemes are theoretically attractive and expressively powerful, but the machinery is heavy and difficult for the novice to deal with, and few empirical studies are available as to the actual performance of these formalisms. Much remains to be done in this area. PDL has showed that the use of precepts can improve learning and generalization. The focus should now be on how to efficiently assist the human teacher by letting the machine either discover precepts on its own or guide the teacher. Questions such as the following will need to be addressed:

- How much information about the world will that machine need?
- What kind of information should it be (i.e., what is common sense)?
- How should that information be represented?
- What learning mechanisms should be used to acquire that information?

APPENDIX A

In this appendix, we summarize the original version of PDLA in which precepts were assumed to be correct and possible exceptions were therefore not dealt with. The results in this Appendix also reflect the fixing of a small bug found in [Giraud-Carrier and Martinez 93]. The variables are as defined in Chapter 3. The network starts empty. For each pair in the training set and for each precept (denoted by *q*), learning goes as follows:

```
Begin  PDLA
Set create_flag to true
Set n to first node in the network (linked list)
While there is an n to execute
 case superset(n.p, q)
      if concordant(n.p, q) then set create_flag = false
 case equal(n.p, q)
      if concordant(n.p, q) then set create_flag = false
      else n self-deletes
 case subset(n.p, q)
      if concordant(n.p, q) then n self-deletes
 case discriminated(n.p, q)
      if concordant(n.p, q) & near_match(n.p, q) &
        num_asserted(n.p)=num_asserted(q) then
           if size(n.p)>=size(q) & num_asserted(n.p)>1
             then
                drop variable in n.p
                set create_flag to false
           else if num_asserted(q)>1 then
                drop variable in q
                n self-deletes
End While
If create_flag then create a new node for q
End  PDLA
```

When new pairs are presented to the network, in the execution phase, each node computes its activation. The node with highest activation wins the competition and the output of its stored pair becomes the network's output. If two nodes have equal activation and one is more general than the other (i.e., different sizes), priority is given to the most general one. If several nodes have the same highest activation and the same size (degree of specificity), priority is arbitrarily given to the first one.

Simulations of this algorithm gave the following results, each averaged over 10 runs in which one (randomly generated) half of the data set was used as the training set and the other half as the test set. The first number is the result of generalization (i.e., the percentage of correct guesses on the test set), the second number is the number of pairs in the training set (including precepts when applicable), and the third number is the size of the network in nodes, after learning.

Application	No precepts			With precepts		
lenses	68.3	12	7	76.7	13	5
zoo	95.6	45	26	95.6	46	22
voting	87.8	217	78	92.6	218	34
iris	91.7	75	42	---		
monk3	77.5	216	120	97.4	218	65
hepatitis	77.1	77	72	79	79	61
mushroom	100	4062	2273	---		

Table A1 - PDLA-Early Version Results

In each case the precepts used were the same as the good precepts described in Chapter 4. In every case, greater parsimony was achieved and generalization was improved.

APPENDIX B

In this appendix, we give the details of the design of PDLA. The design is inspired by object-oriented principles. The software is made up of several modules, each implementing a specific object-class, and providing services to the other modules. All code is written in C++ on a DEC station.

The object class *Vector* implements pairs as dynamic vectors (i.e., 1-dimensional array of dynamic size). It either stores or makes available through computation, all the information relative to pairs (e.g., size, num_asserted).

The object class *Vector_set* implements the data set as a linked list. One of its functions returns a vector chosen at random from the data set. The chosen vector is marked as belonging to the training set. This function allows the generation of a random training set of any arbitrary size. Another function returns the first unmarked vector of the list. This function allows using the remainder of the data set (i.e., vectors not used in training) as a test set. This class is also used to handle the set of precepts. In this case however, all the vectors are left unmarked.

The object class *Node* implements network nodes. Currently, a node contains a vector, a pointer to a next node, and a list of vectors that contradict its own. There is an advantage in having this special class separate from the class Vector. The network's physical implementation can be kept separate from its logical implementation and it is also easy to extend the model so that nodes do or store more than is currently possible (this was in fact useful in going from the early version of Appendix A to the current version). The list of contradicting vectors is used to compute num_exceptions. Before a new vector is placed on the list, it is first checked for redundancy (see Chapter 3). When queried for num_exceptions, the list is traversed and the sizes of overlap of its vectors with the stored vector are added to give the total amount of space the stored vector has been contradicted on.

The object class *Network* implements the network of nodes as a linked list. As mentioned above, the object-oriented design allows it to be easily modified to a tree or any other efficient data structure. This class contains the functions implementing the broadcast and gather facilities. It also deals with possible exceptions (function Process-Exceptions()).

The main program *Pdla* orchestrates the proper interaction of the above object classes. Each data set is processed 10 times (to increase the statistical reliability of the results) and results are averaged accordingly. In each of the ten runs, a new training set is randomly generated and learned, then the test set is presented for generalization.

Half of the data set is used for training and the other half is used for testing.

APPENDIX C

In this appendix, we show the simulation results of the three variants of PDLA discussed in Chapter 5. In each case, the first table has reference to good precepts, while the second has reference to bad precepts. Labels are: % for predictive accuracy (i.e., generalization), TS for the size of the training set including precepts where applicable, and NS for the size of the network after learning.

Priority to More General

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	66.7	12	6	75.8	13	4
zoo	95.8	45	22	95.8	46	19
voting	88.3	217	74	90.9	218	31
iris	87.6	75	33	91.7	81	22
monk3	82.4	216	76	97.4	218	43
hepatitis	77.2	77	71	79.1	79	61

Table C1 - Priority to General-Good Precepts

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	66.7	12	6	50	13	6
zoo	95.8	45	22	75.1	46	23
voting	88.3	217	74	75.7	218	72
iris	87.6	75	33	28.7	81	38
monk3	82.4	216	76	81.9	218	74
hepatitis	77.2	77	71	59.9	79	73

Table C2 - Priority to General-Bad Precepts

Initialize create flag to Network's Output

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	60.8	12	5	77.5	13	4
zoo	95.3	45	9	95.3	46	8
voting	87.3	217	19	90.7	218	13
iris	85.7	75	15	89.9	81	14
monk3	80.8	216	44	87.6	218	27
hepatitis	73.2	77	23	75	79	22

Table C3 - Init create_flag-Good Precepts

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	60.8	12	5	50.8	13	6
zoo	95.3	45	9	84.4	46	12
voting	87.3	217	19	80.6	218	73
iris	85.7	75	15	66.7	81	31
monk3	80.8	216	44	83.3	218	55
hepatitis	73.2	77	23	57.7	79	33

Table C4 - Init create_flag-Bad Precepts

PDLA Based on Current Network's Output

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	60.8	12	5	77.5	13	4
zoo	95.3	45	9	95.3	46	8
voting	84.6	217	21	86.9	218	14
iris	85.7	75	15	89.9	81	14
monk3	80.9	216	44	87.6	218	27
hepatitis	72.9	77	23	75.6	79	22
mushroom	99.9	4062	27	(unnecessary)		

Table C5 - Inductive-Good Precepts

	No precepts			With precepts		
Application	%	TS	NS	%	TS	NS
lenses	60.8	12	5	50.8	13	4
zoo	95.3	45	9	84.4	46	12
voting	84.6	217	21	71.2	218	58
iris	85.7	75	15	66.7	81	31
monk3	80.9	216	44	83.6	218	55
hepatitis	72.9	77	23	58.2	79	34

Table C6 - Inductive-Bad Precepts

BIBLIOGRAPHY

[Barker 93]
Barker, J.C. Eclectic Machine Learning. Ph.D. Dissertation, Brigham Young University, Department of Computer Science, 1993. In preparation.

[Blumer et al. 87]
Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M.K. Occam's Razor. *Information Processing Letters*, *24*, 1987, 377-380. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 201-204.

[Carbonell 83]
Carbonell, J.G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 5.

[Carbonell et al. 83]
Carbonell, J.G., Michalski, R.S., and Mitchell, T.M. An Overview of Machine Learning. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 1.

[Cordier 84]
Cordier, M-O. Les Systemes Experts. *La Recherche*, *15*, 151 (Jan. 1984), 60-69. In French.

[Davis 90]
Davis, E. *Representations of Commonsense Knowledge*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Dietterich and Michalski 83]
Dietterich, T.G., and Michalski, R.S. A Comparative Review of Selected Methods for Learning from Examples. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 3.

[Dreyfus 79]
Dreyfus, H. *What Computers Can't Do: The Limits of Artificial Intelligence*. Revised Edition. Harper & Row, Publishers, 1979.

[Giraud-Carrier and Martinez 93]
Giraud-Carrier, C., and Martinez, T. Precept-Driven Learning: Extending the Training Set Approach. Submitted.

[Haas and Hendrix 83]
Haas, N., and Hendrix, G.G. Learning by Being Told: Acquiring Knowledge for Information Management. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 13.

[Hall and Romaniuk 90]
Hall, L.O., and Romaniuk, S.G. A Hybrid Connectionist, Symbolic Learning System. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990, 783-788.

[Hobbs and Moore 85]
Hobbs, J.R., and Moore, R.C. *Formal Theories of the Commonsense World*. Ablex Publishing Corporation, Norwood, NJ, 1985.

[Hebb 49]
Hebb, D.O. *The Organization of Behavior*. New York: Wiley, 1949.

[Kibler and Aha 87]
Kibler, D., and Aha, D.W. Learning Representative Exemplars of Concepts: An Initial Case Study. In *Proceedings of the Fourth International Workshop on Machine Learning*, 1987, 24-30. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 108-114.

[Lenat et al. 90]
Lenat, D.B., Guha, R.V., Pittman, K., Pratt, D., and Shepherd, M. Cyc: Toward Programs With Common Sense. *Communications of the ACM*, *33*, 8 (Aug. 1990), 30-49.

[Lifschitz 90]
Lifschitz, V. (Ed.). *Formalizing Common Sense: Papers by John McCarthy*. Ablex Publishing Corporation, Norwood, NJ, 1990.

[Martinez 86]
Martinez, T.R. Adaptive Self-Organizing Networks. Ph.D. Dissertation, UCLA Technical Report-CSD 860093, 1986.

[Martinez et al. 93]
Martinez, T., Barker, C., and Giraud-Carrier, C. A Generalizing Adaptive Discriminant Network. Submitted.

[McCarthy 77]

McCarthy, J. Epistemological Problems of Artificial Intelligence. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, 1038–1044.

[McCarthy 80]
McCarthy, J. Circumscription – A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, *13* (1,2), April 1980, 27-39, 171–172.

[McCarthy 84]
McCarthy, J. Some Expert Systems Need Common Sense. In Pagels, H. (Ed.), *Computer Culture: The Scientific, Intellectual and Social Impact of the Computer*. Annals of the New York Academy of Sciences, Vol. 426, 1984, 129–137.

[McCarthy 86]
McCarthy, J. Applications of Circumscription to Formalizing Common Sense Knowledge. *Artificial Intelligence*, *28* (1), 1986, 89–116.

[McCulloch and Pitts 43]
McCulloch, W.S., and Pitts, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, *5*, 1943, 115–133.

[Michalski 83]
Michalski, R.S. A Theory and Methodology of Inductive Learning. *Artificial Intelligence*, *20*, 1983, 111–161. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 70–95.
[Michalski and Larson 78]
Michalski, R.S., and Larson, J.B. Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and Description of Programs ESEL and AQ11. Rept. No. 78-867, Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[Minsky 91]
Minsky, M. Logical Vs. Analogical or Symbolic Vs. Connectionist or Neat Vs. Scruffy. *AI Magazine*, *12*, 2 (Summer 1991), 34–51.

[Minsky and Papert 67]
Minsky, M., and Papert, S. *Perceptrons*. MIT Press, Cambridge, MA, 1967.

[Mitchell 80]
Mitchell, T.M. The Need for Biases in Learning Generalizations. Technical Report No. CBM-TR-117, New Brunswick, NJ: Rutgers University, Department of Computer Science, 1980. Reprinted in Shavlik, J.W., and Dietterich,

T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 184-191.

[Mitchell 82]
Mitchell, T.M. Generalization as Search. *Artificial Intelligence*, *18*, 1982, 203-226. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 96-107.

[Mitchell et al. 83]
Mitchell, T.M., Utgoff, P.E., and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 6. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 510-523.

[Mooney et al. 89]
Mooney, R., Shavlik, J., Towell, G., and Gove, A. An Experimental Comparison of Symbolic and Connectionist Learning Algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989, 775-780. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 171-176.

[Murphy and Aha 92]
Murphy, P.M., and Aha, D.W. *UCI Repository of machine learning databases*. Irvine, CA: University of California, Department of Information and Computer Science, 1992.

[Quinlan 86]
Quinlan, J.R. Induction of Decision Trees. *Machine Learning*, *1*, 1986, 81-106. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 57-69.

[Remy 88]
Remy, C. Quand l'Ordinateur Apprend, *Micro-Systemes*, *88* (Jul-Aug. 1988), 106-118. In French.

[Rosenblatt 58]
Rosenblatt, E. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, *65*, 1958, 386-408. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 138-149.

[Rosenblatt 62]
Rosenblatt, E. *Principles of Neurodynamics*. New York: Spartan, 1962.

[Rumelhart and Zipser 85]
Rumelhart, D.E., and Zipser, D. Feature Discovery by Competitive Learning. *Cognitive Science*, *9*:1, 1985, 75–112. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 307–325.

[Rumelhart et al. 86]
Rumelhart, D.E., Hinton, G.E., and Williams, R.J. Learning Internal Representations by Error Propagation. In Rumelhart, D.E., and McClelland, J.L. (Eds.), *Parallel Distributed Processing* (Vol. I). Cambridge, MA: MIT Press, 1986, Chapter 6. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 115–137.

[Rychener 83]
Rychener, M.D. The Instructible Production System: A Retrospective Analysis. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 14.

[Sejnowski and Rosenberg 87]
Sejnowski, T.J., and Rosenberg, C.R. Parallel Networks that Learn to Pronounce English Text. *Complex Systems*, *1*, 1987, 145–168.

[Shavlik and Dietterich 90]
Shavlik, J.W., and Dietterich, T.G. (Eds.). *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[Shavlik and Towell 89]
Shavlik, J.W., and Towell, G.G. An Approach to Combining Explanation-Based and Neural Learning Algorithms. *Connection Science*, *1*, 1989, 233–255. Reprinted in Shavlik, J.W., and Dietterich, T.G. (Eds.), *Readings in Machine Learning*. Morgan Kaufmann Publishers, Inc., 1990, 828–839.

A Precept-Driven Learning Algorithm

Christophe G. Giraud-Carrier
Department of Computer Science
M.S. Degree, April 1993

ABSTRACT

Machine learning is an attempt at devising mechanisms that machines can use to learn, rather than being explicitly programmed for, real-world applications. The goal of learning systems is to generalize. Generalization is based on the set of critical features the system has available. Training set learners typically extract critical features from a random set of examples drawn from experimentation. This approach can beneficially be extended by endowing the system with some a priori knowledge, in the form of *precepts*. Advantages of the augmented system include speed-up, improved generalization and greater parsimony.

This thesis presents a precept-driven learning algorithm. The main characteristics of the algorithm include: 1) neurally inspired architecture, 2) bounded learning and execution times, and 3) ability to handle both correct and incorrect precepts. Results of simulations on real-world data demonstrate promise.

COMMITTEE APPROVAL: ____________________________________
Tony Martinez, Committee Chair

____
David Embley, Committee Member

____
David Embley, Graduate Coordinator