Efficient Constructive Techniques for Training Switching Neural Networks

Enrico Ferrari and Marco Muselli

Abstract. In this paper a general constructive approach for training neural networks in classification problems is presented. This approach is used to construct a particular connectionist model, named Switching Neural Network (SNN), based on the conversion of the original problem in a Boolean lattice domain. The training of an SNN can be performed through a constructive algorithm, called *Switch Programming* (*SP*), based on the solution of a proper linear programming problem. Since the execution of SP may require excessive computational time, an approximate version of it, named *Approximate Switch Programming* (*ASP*) has been developed. Simulation results obtained on the StatLog benchmark show the good quality of the SNNs trained with SP and ASP.

Keywords: Constructive method, Switching Neural Network, Switch Programming, positive Boolean function synthesis, Statlog benchmark.

1 Introduction

Solving a classification problem consists of finding a function $g(\mathbf{x})$ capable of providing the most probable output in correspondence of any feasible input vector $\mathbf{x}$, when only a finite collection S of examples is available. Since the probability of misclassifying a pattern $\mathbf{x}$ is generally unknown, classification algorithms work by

Enrico Ferrari
Institute of Electronics, Computer and Telecommunication Engineering,
Italian National Research Council, Via De Marini, 6 - 16149 Genoa, Italy
e-mail: ferrari@ieiit.cnr.it

Marco Muselli
Institute of Electronics, Computer and Telecommunication Engineering,
Italian National Research Council, Via De Marini, 6 - 16149 Genoa, Italy
e-mail: muselli@ieiit.cnr.it

L. Franco et al. (Eds.): Constructive Neural Networks, SCI 258, pp. 25–48.
springerlink.com

minimizing at the same time the error on the available data and a measure of the complexity of g. As a matter of fact, according to the Occam razor principle and the main results in statistical learning theory [18] on equal training errors, a simpler function g has a higher probability of scoring a good level of accuracy in examples not belonging to S.

To pursue this double target any technique for the solution of classification problems must perform two different actions: choosing a class Γ of functions (*model definition*) and retrieving the best classifier $g \in \Gamma$ (*training phase*). These two tasks imply a trade-off between a correct description of the data and the generalization ability of the resulting classifier. In fact, if the set Γ is too large, it is likely to incur the problem of *overfitting*: the optimal classifier $g \in \Gamma$ has a good behavior in the examples of the training set, but scores a high number of misclassifications in the other points of the input domain. On the other hand, the choice of a small set Γ prevents retrieval of a function with a sufficient level of accuracy on the training set.

Backpropagation algorithms [17] have been widely used to train multilayer perceptrons: when these learning techniques are applied, the choice of Γ is performed by defining some topological properties of the net, such as the number of hidden layer and neurons. In most cases, this must be done without having any prior information about the problem at hand and several validation trials are needed to find a satisfying network architecture.

In order to avoid this problem, two different approaches have been introduced: pruning methods [16] and constructive techniques [10]. The former consider an initial trained neural network with a large number of neurons and adopt smart techniques to find and eliminate those connections and units which have a negligible influence on the accuracy of the classifier. However, training a large neural network may increase the computational time required to obtain a satisfactory classifier.

On the other hand, constructive methods initially consider a neural network including only the input and the output layers. Then, hidden neurons are added iteratively until a satisfactory description of the examples in the training set is reached. In most cases the connections between hidden and output neurons are decided before training, so that only a small part of the weight matrix has to be updated at each iteration. It has been shown [10] that constructive methods usually present a rapid convergence to a well-generalizing solution and allow also the treatment of complex training sets. Nevertheless, since the inclusion of a new hidden unit involves only a limited number of weights, it is possible that some correlations between the data in the training set may be missed.

Here, we will present a new connectionist model, called *Switching Neural Network (SNN)* [12], which can be trained in a constructive way while achieving generalization errors comparable to those of best machine learning techniques. An SNN includes a first layer containing a particular kind of A/D converter, called *latticizers*, which suitably transform input vectors into binary strings. Then, the subsequent two layers compute a positive Boolean function that solves in a lattice domain the original classification problem.

Since it has been shown [11] that positive Boolean functions can approximate any measurable function within any desired precision, the SNN model is sufficiently

rich to treat any real-world classification problem. A constructive algorithm, called *Switch Programming* (*SP*) has been proposed [3] for training an SNN. It is based on integer linear programming and can lead to an excessive computational burden when a complex training set is analyzed. To allow a wider application of SNN, a suboptimal method, named *Approximate Switch Programming* (*ASP*) will be introduced here. Preliminary results on the Statlog benchmark [9] show that ASP is able to considerably reduce the execution time while keeping a high degree of accuracy in the resulting SNN.

The chapter is organized as follows. In Sec. 2 the considered classification problem is formalized, whereas in Sec. 3 a general schema for a wide class of constructive methods is presented. The SNN model is presented in Sec. 4 and in Sec. 5 the general schema introduced in Sec. 2 is employed to describe the SP and the ASP algorithms.

Sec. 6 shows how it is possible obtain a set of intelligible rules starting from any trained SNN, whereas Sec. 7 illustrates a very simple example with the purpose of making clear the functioning of an SNN. Finally, Sec. 8 presents the good results obtained with SP and ASP algorithms on the well-known datasets of the Statlog benchmark. Some concluding remarks end the chapter.

2 Problem Setting

Consider a general binary classification problem, where d-dimensional patterns $\mathbf{x}$ are to be assigned to one of two possible classes, labeled with the values of a Boolean output variable $y \in \{0,1\}$. According to possible situations in real world problems, the type of the components x_i, $i = 1,\ldots,d$, may be one of the following:

- **continuous ordered**: when x_i can assume values inside an uncountable subset U_i of the real domain $\mathbb{R}$; typically, U_i is an interval $[a_i, b_i]$ (possibly open at one end or at both the extremes) or the whole $\mathbb{R}$.
- **discrete ordered**: when x_i can assume values inside a countable set C_i, where a total ordering is defined; typically, C_i is a finite subset of $\mathbb{Z}$.
- **nominal**: when x_i can assume values inside a finite set H_i, where no ordering is defined; for example, H_i can be a set of colors or a collection of geometric shapes.

If x_i is a binary component, it can be viewed as a particular case of discrete ordered variable or as a nominal variable; to remove a possible source of ambiguity, a binary component will always be considered henceforth as a nominal variable.

Denote with I_m the set $\{1,2,\ldots,m\}$ of the first m positive integers; when the domain C_i of a discrete ordered variable is finite, it is isomorphic to I_m, with $m = |C_i|$, being $|A|$ the cardinality of the set A. On the other hand, if C_i is infinite, it can be shown to be isomorphic to the set $\mathbb{Z}$ of the integer numbers (if C_i is neither lower nor upper bounded), to the set $\mathbb{N}$ of the positive integers (if C_i is lower bounded), or to the set $\mathbb{Z} \setminus \mathbb{N}$ (if C_i is upper bounded).

For each of the above three types of components x_i a proper metric can be defined. In fact, many different distances have been proposed in the literature to characterize the subsets of $\mathbb{R}$. Also when the standard topology is assumed, different equivalent metrics can be adopted; throughout this paper, the *absolute metric* $d_a(v,v') = |v-v'|$ will be employed to measure the dissimilarity between two values $v, v' \in \mathbb{R}$ assumed by a continuous ordered component x_i.

In the same way, when x_i is a discrete ordered component, the above cited isomorphism between its domain C_i and a suitable subset K of $\mathbb{Z}$ makes it possible to adopt the distance $d_c(v,v')$ induced on C_i by the absolute metric $d_a(v,v') = |v-v'|$ on K. Henceforth we will use the term *counter metric* to denote the distance d_c.

Finally, if x_i is a nominal component no ordering relation exists between any pair of elements of its domain H_i; we can only assert that a value v assumed by x_i is equal or not to another value v'. Consequently, we can adopt in H_i the *flat metric* $d_f(v,v')$ defined as

$$d_f(v,v') = \begin{cases} 0 \text{ if } v = v' \\ 1 \text{ if } v \neq v' \end{cases}$$

for every $v, v' \in H_i$. Note that, by considering a counting function η which assigns a different positive integer in I_m to each element of the set H_i, being $m = |H_i|$, we can substitute the domain H_i of x_i with the set I_m without affecting the given classification problem. It is sufficient to employ the flat metric $d_f(v,v')$ also in I_m, which is no longer seen as an ordered set.

According to this framework, to simplify the exposition we suppose henceforth that the patterns $\mathbf{x}$ of our classification problem belong to a set $X = \prod_{i=1}^{d} X_i$, where each monodimensional domain X_i can be a subset of the real field $\mathbb{R}$ if x_i is a continuous ordered variable, a subset of integers in $\mathbb{Z}$ if x_i is a discrete ordered component, or the finite set I_m (for some positive integer m) without ordering on it if x_i is a nominal variable.

A proper metric $d_X(\mathbf{x},\mathbf{x}')$ on X can be simply defined by summing up the contributions $d_i(x_i,x_i')$ given by the different components

$$d_X(\mathbf{x},\mathbf{x}') = \sum_{i=1}^{d} d_i(x_i,x_i') \,, \quad \text{for any } \mathbf{x},\mathbf{x}' \in X$$

where d_i is the absolute metric d_a if x_i and x_i' are (continuous or discrete) ordered variables or the flat metric d_f if x_i and x_i' are nominal variables.

The target of a binary classification problem is to choose within a predetermined set Γ of decision functions the classifier $g : X \to \{0,1\}$ that minimizes the number of misclassifications on the whole set X. If Γ is equal to the collection $\mathscr{M}$ of all the measurable decision functions, this amounts to selecting the *Bayes classifier* $g_{\text{opt}}(\mathbf{x})$ [2]. On the other hand, if Γ is a proper subset of $\mathscr{M}$, the optimal decision function corresponds to the classifier $g^* \in \Gamma$ that best approximates g_{opt} according to a proper distance in $\mathscr{M}$.

Unfortunately, in real world problems we have only access to a *training set* S, i.e. a collection of s observations $(\mathbf{x}_k, y_k)$, $k = 1,\ldots,s$, for the problem at hand.

Thus, the solution to the binary classification problem is produced by adopting a *learning algorithm* $\mathscr{A}$ that employs the information contained in the training set to retrieve the best classifier g^* in Γ or a good approximation $\hat{g}$ to it.

This approach consists therefore of two different stages:

1. at first the class Γ of decision functions must be suitably determined (*model selection*);
2. then, the best classifier $g^* \in \Gamma$ (or a good approximation $\hat{g}$) is retrieved through the learning algorithm $\mathscr{A}$ (*training phase*).

In the next section we will introduce a general constructive model, which is sufficiently rich to approximate within an arbitrary precision any measurable function $g : X \to \{0,1\}$.

3 A General Structure for a Class of Constructive Methods

Denote with $S_1 = \{\mathbf{x}_k \mid y_k = 1\}$ the set of positive examples in the training set S and with $S_0 = \{\mathbf{x} \mid \ y_k = 0\}$ the set of negative examples. Moreover, let $s_1 = |S_1|$ and $s_0 = |S_0|$.

In many constructive methods the function $\hat{g}$ is realized by a two layer neural network; the hidden layer is built incrementally by adding a neuron at each iteration of the training procedure. In order to characterize the hidden neurons consider the following

Definition 1. A collection $\{\{L_h, \hat{y}_h\},\ h = 1,\ldots,t+1\}$, where $L_h \subset X$ and $\hat{y}_h \in \{0,1\}$ for each $h = 1,\ldots,t$, will be called a *decision list* for a two class problem if $L_{t+1} = X$.

In [10] the decision list is used hierarchically: a pattern $\mathbf{x}$ is assigned to the class y_h, where h is the lower index such that $\mathbf{x} \in L_h$. It is possible to consider more general criteria in the output assignment: for example a weight $w_h > 0$ can be associated with each domain L_h, measuring the reliability of assigning the output value $\hat{y}_h$ to every point in L_h.

It is thus possible to associate with every pattern $\mathbf{x}$ a weight vector $\mathbf{u}$, whose h-th component is defined by

$$u_h = \begin{cases} w_h & \text{if } \mathbf{x} \in L_h \\ 0 & \text{otherwise} \end{cases}$$

for $h = 1,\ldots,t$. The weight u_h can be used to choose the output for the pattern $\mathbf{x}$. Without loss of generality suppose that the decision list is ordered so that $\hat{y}_h = 0$ for $h = 1,\ldots,t_0$, whereas $\hat{y}_h = 1$ for $h = t_0+1,\ldots,t_0+t_1$, where $t_0 + t_1 = t$. The value of $\hat{y}_{t+1}$ is the *default* decision, i.e. the output assigned to $\mathbf{x}$ if $\mathbf{x} \notin L_h$ for each $h = 1,\ldots,t$.

In order to fix a criterion in the output assignment for an input vector $\mathbf{x}$ let us present the following

Definition 2. A function $\sigma(\mathbf{u}) \in \{0,1\}$ is called an *output decider* if

$$\sigma(\mathbf{u}) = \begin{cases} y_{t+1} & \text{if } u_1 = \ldots = u_{t_0} = u_{t_0+1} = \ldots = u_t = 0 \\ 1 & \text{if } u_1 = \ldots = u_{t_0} = 0 \text{ and some } u_h > 0 \text{ with } t_0 < h \leq t \\ 0 & \text{if } u_{t_0+1} = \ldots = u_t = 0 \text{ and some } u_h > 0 \text{ with } 0 < h \leq t_0 \end{cases}$$

This classifier can then be implemented in a two layer neural network: the first layer retrieves the weights u_h for $h = 1,\ldots,t$, whereas the second one realizes the output decider σ. The behavior of σ is usually chosen *a priori* so that the training phase consists of finding a proper decision list and the relative weight vector $\mathbf{w}$. For example, σ can be made equivalent to a comparison between the sum of the weights of the two classes:

$$\sigma(\mathbf{u}) = \begin{cases} 0 & \text{if } \sum_{h=1}^{t_0} u_h > \sum_{h=t_0+1}^{t} u_h \\ 1 & \text{if } \sum_{h=1}^{t_0} u_h < \sum_{h=t_0+1}^{t} u_h \\ \hat{y}_{t+1} & \text{otherwise} \end{cases}$$

The determination of the decision list $\{L_h, \hat{y}_h\}$, $h = 1,\ldots,t$, can be performed in a constructive way, by adding at each iteration h the best pair $\{L_h, \hat{y}_h\}$ according to a smart criterion. Each domain L_h corresponds to a neuron characterized through the function introduced by the following

Definition 3. Consider a subset $T \subset S_y$, $y \in \{0,1\}$. The function

$$\hat{g}_h(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in L_h \\ 0 & \text{otherwise} \end{cases}$$

is called a *partial classifier* for T if $T \cap L_h$ is not empty whereas $L_h \cap \setminus S_{1-y} = \emptyset$. If $\hat{g}_h(\mathbf{x}) = 1$ the h-th neuron will be said to *cover* $\mathbf{x}$.

The presence of noisy data can also be taken into account by allowing a small number of errors in the training set. To this aim Def. 3 can be generalized by the following

Definition 4. Consider a subset $T \subset S_y$, $y \in \{0,1\}$. The function

$$\hat{g}_h(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in L_h \\ 0 & \text{otherwise} \end{cases}$$

is called a *partial classifier with error* ε for T if $T \cap L_h$ is not empty whereas $\left|L_h \cap S_{1-y}\right| \leq \varepsilon |S_{1-y}|$.

It is easy to notice that Def. 3 is recovered when setting $\varepsilon = 0$.

Since the target of the training phase is to find the simplest network satisfying the input-output relations in the training set, the patterns already covered by at least one neuron can be ignored when training further neurons having the same value of $\hat{y}_h$.

Fig. 1 shows a general constructive procedure for training a neural network in the case of binary output. At each iteration the set T contains the patterns belonging to

Constructive training for a two layer perceptron

For $y \in \{0,1\}$ do

1. Set $T = S_y$ and $h = 1$.
2. Find a partial classifier $\hat{g}_h$ for T.
3. Let $W = \{\{\mathbf{x}_k, y_k\} \in T \mid \hat{g}_h(\mathbf{x}_k) = 1\}$.
4. Set $T = T \setminus W$ and $h = h + 1$.
5. If T is nonempty go to step 2.
6. Prune redundant neurons and set $t_y = h$.

Fig. 1 General constructive procedure followed for neural network training.

the current output value not covered by the neurons already included in the network. Notice that removing elements from T allows a considerable reduction of the training time for each neuron since a lower number of examples has to be processed at each iteration.

A pruning phase is performed at the end of the training process in order to eliminate redundant overlaps among the sets L_h, $h = 1, \ldots, t$.

Without entering into details about the general theoretical properties of constructive techniques, which can be found in [10], in the following sections we will present the architecture of Switching Neural Networks and an appropriate training algorithm.

4 Switching Neural Networks

A promising connectionist model, called Switching Neural Network (SNN), has been developed recently[12]. According to this model, the input variables are transformed into n-dimensional Boolean strings by means of a particular mapping $\varphi : X \to \{0,1\}^n$, called *latticizer*.

Consider the Boolean lattice $\{0,1\}^n$, equipped with the well known binary operations '$+$' (*logical sum* or OR) and '$\cdot$' (*logical product* or AND). To improve readability, the elements of this Boolean lattice will be denoted henceforth as strings of bits: in this way, the element $(0,1,1,0) \in \{0,1\}^4$ will be written as 0110. The usual priority on the execution of the operators $+$ and $\cdot$ will be adopted; furthermore, when there is no possibility of misleading, the symbol $\cdot$ will be omitted, thus writing $\mathbf{v}\mathbf{v}'$ instead of $\mathbf{v} \cdot \mathbf{v}'$.

A standard partial ordering on $\{0,1\}^n$ can be defined by setting $\mathbf{v} \leq \mathbf{v}'$ if and only if $\mathbf{v} + \mathbf{v}' = \mathbf{v}'$; this definition is equivalent to writing $\mathbf{v} \leq \mathbf{v}'$ if and only if $v_i \leq v'_i$ for every $i = 1, \ldots, n$. According to this ordering, a Boolean function

$f : \{0,1\}^n \rightarrow \{0,1\}$ is called *positive* (resp. *negative*) if $\mathbf{v} \leq \mathbf{v}'$ implies $f(\mathbf{v}) \leq f(\mathbf{v}')$ (resp. $f(\mathbf{v}) \geq f(\mathbf{v}')$) for every $\mathbf{v}, \mathbf{v}' \in \{0,1\}^n$. Positive and negative Boolean functions form the class of *monotone Boolean functions*.

Since the Boolean lattice $\{0,1\}^n$ does not involve the complement operator NOT, Boolean expressions developed in this lattice (sometimes called *lattice expressions*) can only include the logical operations AND and OR. As a consequence, not every Boolean function can be written as a lattice expression. It can be shown that only positive Boolean functions are allowed to be put in the form of lattice expressions.

A recent theoretical result [11] asserts that positive Boolean functions are universal approximators, i.e. they can approximate every measurable function $g : X \rightarrow \{0,1\}$, being X the domain of a general binary classification problem, as defined in Sec. 2. Denote with Q_n^l the subset of $\{0,1\}^n$ containing the strings of n bits having exactly l values 1 inside them. A possible procedure for finding the positive Boolean function f that approximates a given g within a desired precision is based on the following three steps:

1. (*Discretization*) For every ordered input x_i, determine a finite partition $\mathscr{B}_i$ of the domain X_i such that a function $\hat{g}$ can be found, which approximates g on X within the desired precision and assumes a constant value on every set $B \in \mathscr{B}$, where $\mathscr{B} = \{\prod_{i=1}^d B_i : B_i \in \mathscr{B}_i, i = 1, \ldots, d\}$.
2. (*Latticization*) By employing a proper function φ, map the points of the domain X into the strings of Q_n^l, so that $\varphi(\mathbf{x}) = \varphi(\mathbf{x}')$ if $\mathbf{x}$ and $\mathbf{x}'$ belong to the same set $B \in \mathscr{B}$, whereas $\varphi(\mathbf{x}) \neq \varphi(\mathbf{x}')$ if $\mathbf{x} \in B$ and $\mathbf{x}' \in B'$, being B and B' two different sets in $\mathscr{B}$.
3. (*Positive Boolean function synthesis*) Select a positive Boolean function f.

If g is completely known these three steps can be easily performed; the higher the required precision is, the finer the partitions $\mathscr{B}_i$ for the domains X_i must be. This affects the length n of the binary strings in Q_n^l, which has to be big enough to allow the definition of the 1-1 mapping φ.

If $\mathbf{a} \in \{0,1\}^n$, let $P(\mathbf{a})$ be the subset of $I_n = \{1, \ldots, n\}$ including the indexes i for which $a_i = 1$. It can be shown [14] that a positive Boolean function can always be written as

$$f(\mathbf{z}) = \bigvee_{\mathbf{a} \in A} \bigwedge_{j \in P(\mathbf{a})} z_j \tag{1}$$

where A is an *antichain* of the Boolean lattice $\{0,1\}^n$, i.e. a set of Boolean strings such that neither $\mathbf{a} < \mathbf{a}'$ nor $\mathbf{a}' < \mathbf{a}$ holds for each $\mathbf{a}, \mathbf{a}' \in A$. It can be proved that a positive Boolean function is univocally specified by the antichain A, so that the task of retrieving f can be transformed into searching for a collection A of strings such that $\mathbf{a}' < \mathbf{a}$ for each $\mathbf{a}, \mathbf{a}' \in A$.

The symbol $\bigvee$ (resp. $\bigwedge$) in (1) denotes a logical sum (resp. product) among the terms identified by the subscript. The logical product $\bigwedge_{j \in P(\mathbf{a})} z_j$ is an *implicant* for the function f; however, when no confusion arises, the term implicant will also be used to denote the corresponding binary string $\mathbf{a} \in A$.

Preliminary tests have shown that a more robust method for classification problems consists of defining a positive Boolean function f_y (i.e. an antichain A_y to be

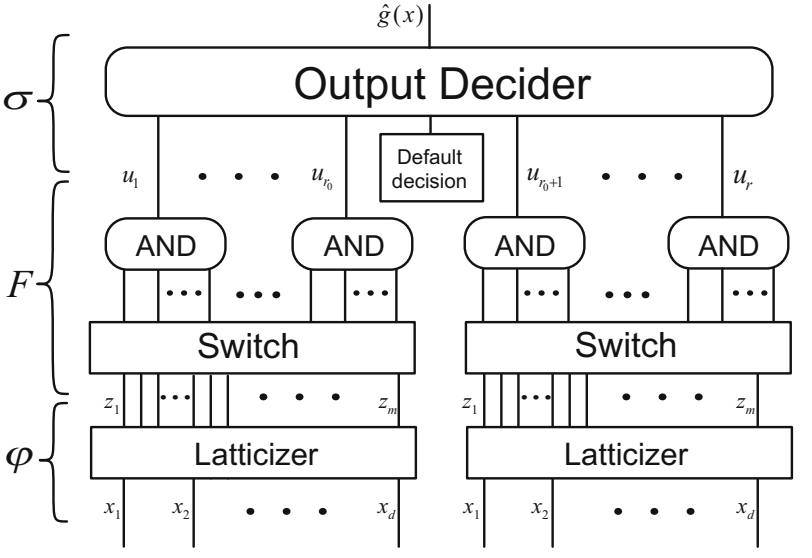

Fig. 2 The schema of a Switching Neural Network

inserted on (1)) for each output value y and in properly combining the functions relative to the different output classes. To this aim, each generated implicant can be characterized by a weight $w_h > 0$, which measures its significance level for the examples in the training set. Thus, to each Boolean string $\mathbf{z}$ can be assigned a weight vector $\mathbf{u}$ whose h-th component is

$$u_h = F_h(\mathbf{z}) = \begin{cases} w_h & \text{if } \bigwedge_{j \in P(\mathbf{a}_h)} z_j = 1 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{a}_h \in A_0 \cup A_1$ for $h = 1, \ldots, t$.

At the final step of the classification process, an output decider $\sigma(\mathbf{u})$ assigns the correct class to the pattern $\mathbf{z}$ according to a comparison between the weights u_h of the different classes. If no h exists such that $u_h > 0$, the default output is assigned to $\mathbf{z}$.

The device implementing the function $\hat{g}(\mathbf{x}) = \sigma(\mathbf{F}(\varphi(\mathbf{x})))$ is shown in Fig. 2. It can be considered a three layer feedforward neural network. The first layer is responsible for the latticization mapping φ; the second realizes the function $\mathbf{F}$ assigning a weight to each implicant. Finally, the third layer uses the weight vector $\mathbf{u} = \mathbf{F}(\mathbf{z})$ to decide the output value for the pattern $\mathbf{x}$.

Every AND port in the second layer is connected only to some of the outputs leaving the latticizers; they correspond to values 1 in the associated implicant. The choice of such values is performed by a *switch* port. For this reason the connectionist model shown in Fig. 2 is called *Switching Neural Network*.

Notice that the device can be subdivided into two parts: the left part includes the t_0 neurons characterizing the examples having output $y = 0$, whereas the right part involves the $t_1 = t - t_0$ implicants relative to the output $y = 1$. For this reason,

the generalization of an SNN to a multiclass problem (where $y \in \{1,\ldots,c\}$, $c > 2$) is immediate: it is sufficient to create a set of implicants for each output value. In addition, the default decision has to be transformed into a *default decision list*, so that if two (or more) output values score the same level of confidence (according to the criterion fixed by σ), the device selects the one with a higher rank in the list.

It is interesting to observe that, unlike standard neural networks, SNNs do not involve floating point operations. In fact the weights $\mathbf{w}$ are provided by the training process and can be chosen from a set of integer values. Moreover, the antichain A can be converted into a set of intelligible rules in the form

$$\textbf{if } < \textit{premise} > \textbf{ then } < \textit{consequence} >$$

through the application of a smart inverse operator [12] of φ to the elements of A.

4.1 Discretization

Since the exact behavior of the function g is not known, the approximating function $\hat{g}$ and the partition $\mathscr{B}$ have to be inferred from the samples $(\mathbf{x}_k, y_k) \in S$. It follows that at the end of the discretization task every set $B_i \in \mathscr{B}_i$ must be large enough to include the component x_{ki} of some point $\mathbf{x}_k$ in the training set. Nevertheless, the resulting partition $\mathscr{B}$ must be fine enough to capture the actual complexity of the function g.

Several different discretization methods for binary classification problems have been proposed in the literature [1, 5, 6, 7]. Usually, for each ordered input x_i a set of $m_i - 1$ consecutive values $r_{i1} < r_{i2} < \cdots < r_{i,m_i-1}$ is generated and the partition $\mathscr{B}_i$ is formed by the m_i sets $X_i \cap R_{ij}$, where $R_{i1} = (-\infty, r_{i1})$, $R_{i2} = (r_{i1}, r_{i2})$, $\ldots$, $R_{i,m_i-1} = (r_{i,m_i-2}, r_{i,m_i-1})$, $R_{im_i} = (r_{i,m_i-1}, +\infty)$. Excellent results have been obtained with the algorithms ChiMerge and Chi2 [5, 7], which employ the χ^2 statistic to decide the position of the points r_{ij}, $k = 1,\ldots,m_i - 1$, and with the technique Ent-MDL [6], which adopts entropy estimates to achieve the same goal. An alternative and promising approach is offered by the method used in the LAD system [1]: in this case an integer programming problem is solved to obtain optimal values for the cutoffs r_{ij}.

By applying a procedure of this kind, the discretization task defines for each ordered input x_i a mapping $\psi_i : X_i \to I_{m_i}$, where $\psi_i(z) = j$ if and only if $z \in R_{ij}$. If we assume that ψ_i is the identity function with $m_i = |X_i|$ when x_i is a nominal variable, the approximating function $\hat{g}$ is uniquely determined by a discrete function $h : I \to \{0,1\}$, defined by $h(\psi(\mathbf{x})) = \hat{g}(\mathbf{x})$, where $I = \prod_{i=1}^{d} I_{m_i}$ and $\psi(\mathbf{x})$ is the mapping from X to I, whose ith component is given by $\psi_i(x_i)$.

By definition, the usual ordering relation is induced by ψ_i on I_{m_i} when x_i is an ordered variable. On the other hand, since in general ψ_i is not 1-1, different choices for the metric on I_{m_i} are possible. For example, if the actual distances on X_i must be taken into account, the metric $d_i(j,k) = |r_{ij} - r_{ik}|$ can be adopted for any $j,k \in I_{m_i}$,

having set $r_{i,m_i} = 2r_{i,m_i} - r_{i,m_i-1}$. Alternative definitions employ the mean points of the intervals R_{ik} or their lower boundaries.

According to statistical non parametric inference methods a valid choice can also be to use the absolute metric d_a on I_{m_i}, without caring about the actual value of the distances on X_i. This choice assumes that the discretization method has selected correctly the cutoffs r_{ij}, sampling with greater density the regions of X_i where the unknown function g changes more rapidly. In this way the metric d on $I = \prod_{i=1}^{d} I_{m_i}$ is given by

$$d(\mathbf{v}, \mathbf{v}') = \sum_{i=1}^{d} d_i(u_i, v_i)$$

where d_i is the absolute metric d_a (resp. the flat metric d_f) if x_i is an ordered (resp. nominal) input.

4.2 Latticization

It can be easily observed that the function ψ provides a mapping from the domain X onto the set $I = \prod_{i=1}^{d} I_{m_i}$, such that $\psi(\mathbf{x}) = \psi(\mathbf{x}')$ if $\mathbf{x}$ and $\mathbf{x}'$ belong to the same set $B \in \mathscr{B}$, whereas $\psi(\mathbf{x}) \neq \psi(\mathbf{x}')$ if $\mathbf{x} \in B$ and $\mathbf{x}' \in B'$, being B and B' two different sets in $\mathscr{B}$. Consequently, the 1-1 function φ from X to Q_n^l, required in the latticization step, can be simply determined by defining a proper 1-1 function β that maps the elements of I into the binary strings of Q_n^l. In this way, $\varphi(\mathbf{x}) = \beta(\psi(\mathbf{x}))$ for every $\mathbf{x} \in X$.

A possible way of constructing the function β is to define properly d mappings $\beta_i : I_{m_i} \to Q_{n_i}^{l_i}$; then, the binary string $\beta(\mathbf{v})$ for an integer vector $\mathbf{v} \in I$ is obtained by concatenating the strings $\beta_i(v_i)$ for $i = 1, \ldots, d$. With this approach, $\beta(\mathbf{v})$ always produces a binary string with length $n = \sum_{i=1}^{d} n_i$ having $l = \sum_{i=1}^{d} l_i$ values 1 inside it.

The mappings β_i can be built in a variety of different ways; however, it is important that they fulfill the following two basic constraints in order to simplify the generation of an approximating function $\hat{g}$ that generalizes well:

1. β_i must be an *isometry*, i.e. $D_i(\beta_i(v_i), \beta_i(v_i')) = d_i(v_i, v_i')$, where $D_i(\cdot,\cdot)$ is the metric adopted on $Q_{n_i}^{l_i}$ and $d_i(\cdot,\cdot)$ is the distance on I_{m_i} (the absolute or the flat metric depending on the type of the variable x_i),
2. if x_i is an ordered input, β_i must be *full order-preserving*, i.e. $\beta_i(v_i) \preceq \beta_i(v_i')$ if and only if $v_i \leq v_i'$, where $\prec$ is a (partial or total) ordering on $Q_{n_i}^{l_i}$.

A valid choice for the definition of $\prec$ consists of adopting the *lexicographic ordering* on $Q_{n_i}^{l_i}$, which amounts to asserting that $\mathbf{z} \prec \mathbf{z}'$ if and only if $z_k < z_k'$ for some $k = 1, \ldots, n_i$ and $z_i = z_i'$ for every $i = 1, \ldots, k-1$. In this way $\prec$ is a total ordering on $Q_{n_i}^{l_i}$ and it can be easily seen that $Q_{n_i}^{l_i}$ becomes isomorphic to I_m with $m = \binom{n_i}{l_i}$. As a consequence the counter metric d_c can be induced on $Q_{n_i}^{l_i}$; this will be the definition for the distance D_i when x_i is an ordered input.

Note that if $l_i = n_i - 1$, binary strings in $Q_{n_i}^{l_i}$ contain a single value 0. Let us suppose that two elements $\mathbf{z}$ and $\mathbf{z}' \in Q_{n_i}^{n_i-1}$ have the value 0 at the jth and at the j' position, respectively; then, we have $\mathbf{z} \prec \mathbf{z}'$ if and only if $j < j'$. Moreover, the distance $D_i(\mathbf{z}, \mathbf{z}')$ between $\mathbf{z}$ and $\mathbf{z}'$ is simply given by the absolute difference $|k - k'|$. As an example, consider for $n_i = 6$ the strings $\mathbf{z} = 101111$ and $\mathbf{z}' = 111101$, belonging to Q_6^5. The application of the above definitions gives $\mathbf{z} \prec \mathbf{z}'$ and $D_i(\mathbf{z}, \mathbf{z}') = 3$, since the value 0 is at the 2nd place in $\mathbf{z}$ and at the 5th place in $\mathbf{z}'$.

If x_i is a nominal variable, the flat metric can also be adopted for the elements of $Q_{n_i}^{l_i}$, thus obtaining

$$D_i(\mathbf{z}, \mathbf{z}') = \begin{cases} 0 \text{ if } \mathbf{z} = \mathbf{z}' \\ 1 \text{ if } \mathbf{z} \neq \mathbf{z}' \end{cases}, \quad \text{for every } \mathbf{z}, \mathbf{z}' \in Q_{n_i}^{l_i}$$

With these definitions, a mapping β_i that satisfies the two above properties (isometry and full order-preserving) is the *inverse only-one code*, which maps an integer $v_i \in I_{m_i}$ into the binary string $\mathbf{z}_i \in Q_{m_i}^{m_i-1}$ having length m_i and jth component z_{ij} given by

$$z_{ij} = \begin{cases} 0 \text{ if } v_i = j \\ 1 \text{ otherwise} \end{cases}, \quad \text{for every } j = 1, \ldots, m_i$$

For example, if $m_i = 6$ we have $\beta_i(2) = 101111$ and $\beta_i(5) = 111101$.

It can be easily seen that the function β, obtained by concatenating the d binary strings produced by the components β_i, maps the integer vectors of I into the set Q_m^{m-d}, being $m = \sum_{i=1}^{d} m_i$. If the metric $D(\mathbf{z}, \mathbf{z}') = \sum_{i=1}^{d} D_i(\mathbf{z}_i, \mathbf{z}_i')$ is employed on Q_m^{m-d}, where $\mathbf{z}_i$ is the binary string formed by the m_i bits of $\mathbf{z}$ determined by I_{m_i} through β_i, we obtain that the 1-1 mapping β is an isometry.

The behavior of the mapping β allows us to retrieve a convenient form for the 1-1 function φ from X to Q_n^l, to be introduced in the latticization step, if the discretization task has produced for each ordered input x_i a set of $m_i - 1$ cutoffs r_{ij}, as described in the previous subsection. Again, let $\varphi(\mathbf{x})$ be obtained by the concatenation of d binary strings $\varphi_i(x_i)$ in $Q_{m_i}^{m_i-1}$. To ensure that $\varphi(\mathbf{x}) = \beta(\psi(\mathbf{x}))$, it is sufficient to define the jth bit z_{ij} of $\mathbf{z}_i = \varphi_i(x_i)$ as

$$z_{ij} = \begin{cases} 0 \text{ if } x_i \in R_{ij} \\ 1 \text{ otherwise} \end{cases}, \quad \text{for every } j = 1, \ldots, m_i \tag{2}$$

if x_i is an ordered variable and as

$$z_{ij} = \begin{cases} 0 \text{ if } x_i = j \\ 1 \text{ otherwise} \end{cases}, \quad \text{for every } j = 1, \ldots, m_i \tag{3}$$

if x_i is a nominal input. Note that $x_i \in R_{ij}$ if and only if x_i exceeds the cutoff $r_{i,j-1}$ (if $j > 1$) and is lower than the subsequent cutoff r_{ij} (if $j < m_i$).

Consequently, the mapping φ_i can be implemented by a simple device that receives in input the value x_i and compares it with a sequence of integers or real numbers, according to definitions (2) or (3), depending on whether x_i is an ordered

or a nominal input. This device will be called *latticizer*; it produces m_i binary outputs, but only one of them can assume the value 0. The whole mapping φ is realized by a parallel of d latticizer, each of which is associated with a different input x_i.

5 Training Algorithm

Many algorithms are present in literature to reconstruct a Boolean function starting from a portion of its truth table. However two drawbacks prevent the use of such techniques for the current purpose: these methods usually deal with general Boolean functions and not with positive ones and they lack generalization ability. In fact, the aim of most of these algorithms is to find a minimal set of implicants which satisfies all the known input-output relations in the truth table. However, for classification purposes, it is important to take into account the behavior of the generated function on examples not belonging to the training set. For this reason some techniques [1, 4, 13] have been developed in order to maximize the generalization ability of the resulting standard Boolean function. On the other hand, only one method, named *Shadow Clustering* [14], is expressly devoted to the reconstruction of positive Boolean functions.

In this section a novel constructive algorithm for building a single f_y (denoted only by f for simplicity) will be described. The procedure must be repeated for each value of the output y in order to find an optimal classifier for the problem at hand. In particular, if the function f_y is built, the Boolean output 1 will be assigned to the examples belonging to the class y, whereas the Boolean output 0 will be assigned to all the remaining examples.

The architecture of the SNN has to be constructed starting from the converted training set S', containing s_1 positive examples and s_0 negative examples. Let us suppose, without loss of generality, that the set S' is ordered so that the first s_1 examples are positive. Since the training algorithm sets up, for each output value, the switches in the second layer of the SNN, the constructive procedure of adding neurons step by step will be called *Switch Programming* (*SP*).

5.1 Implicant Generation

When a Boolean string $\mathbf{z}$ is presented as input, the output of the logical product $\bigwedge_{j \in P(\mathbf{a})} z_j$ at a neuron is positive if and only if $\mathbf{a} \leq \mathbf{z}$ according to the standard ordering in the Boolean lattice. In this case $\mathbf{a}$ will be said to *cover* $\mathbf{z}$.

The aim of a training algorithm for an SNN is to find the simplest antichain A covering all the positive examples and no negative examples in the training set. This target will be reached in two steps: first an antichain A' is generated, then redundant elements of A' are eliminated thus obtaining the final antichain A. A constructive approach for constructing A' consists of generating implicants one at a time according to a smart criterion of choice determined by an objective function $\Phi(\mathbf{a})$.

In particular $\Phi(\mathbf{a})$ must take into account the number of examples in S' covered by $\mathbf{a}$ and the degree of complexity of $\mathbf{a}$, usually defined as the number of elements in $P(\mathbf{a})$ or, equivalently, as the sum $\sum_{i=1}^{m} a_i$. These parameters will be balanced in the objective function through the definition of two weights λ and μ.

In order to define the constraints to the problem, define, for each example $\mathbf{z}_k$, the number ξ_k of indexes i for which $a_i = 1$ and $z_{ki} = 0$. It is easy to show that $\mathbf{a}$ covers $\mathbf{z}_k$ if and only if $\xi_k = 0$. Then, the quantity

$$\sum_{k=1}^{s_1} \theta(\xi_k)$$

where θ represents the usual Heaviside function (defined by $\theta(u) = 1$ if $u > 0$, $\theta(u) = 0$ otherwise), is the number of positive patterns not covered by $\mathbf{a}$. However, it is necessary that $\xi_k > 0$ for each $k = s_1 + 1, \ldots, s$, so that any negative pattern is not covered by $\mathbf{a}$.

Starting from these considerations, the best implicant can be retrieved by solving the following optimization problem:

$$\begin{aligned} \min_{\xi,\mathbf{a}} \ & \frac{\lambda}{s_1} \sum_{k=1}^{s_1} \xi_k + \frac{\mu}{m} \sum_{i=1}^{m} a_i \\ \text{subj to } & \sum_{i=1}^{m} a_i (a_i - z_{ki}) = \xi_k \quad \text{for } k = 1, \ldots, s_1 \\ & \sum_{i=1}^{m} a_i (a_i - z_{ki}) \geq 1 \quad \text{for } k = s_1 + 1, \ldots, s \\ & \xi_k \geq 0 \quad \text{for } k = 1, \ldots, s_1 \\ & a_i \in \{0, 1\} \quad \text{for } i = 1, \ldots, d \end{aligned} \tag{4}$$

where the Heaviside function has been substituted by its argument in order to avoid nonlinearity in the cost function. Notice that the terms in the objective function are normalized in order to be independent of the complexity of the problem at hand.

Since the determination of a sufficiently great collection of implicants, from which the antichain A is selected, requires the repeated solution of problem (4), the generation of an already found implicant must be avoided at any extraction. This can be obtained by adding the following constraint

$$\sum_{i=1}^{m} a_{ji} (1 - a_i) \geq 1 \quad \text{for } j = 1, \ldots, q-1 \tag{5}$$

where $\mathbf{a}$ is the implicant to be constructed and $\mathbf{a}_1, \ldots, \mathbf{a}_{q-1}$ are the already found $q - 1$ implicants.

Additional requirements can be added to problem (4) in order to improve the quality of the implicant $\mathbf{a}$ and the convergence speed. For example, in order to better differentiate implicants and to cover all the patterns in fewer steps, the set S_1'' of

positive patterns not yet covered can be considered separately and weighted by a different factor $\nu \neq \lambda$.

Moreover, in the presence of noise it would be useful to avoid excessive adherence of $\mathbf{a}$ with the training set by accepting a small fraction ε of errors.

In this case a further term is added to the objective function, measuring the level of misclassification, and constraints in (4) have to be modified in order to allow at most εs_0 patterns to be misclassified by the implicant $\mathbf{a}$. In particular, slack variables ξ_k, $k = s_1+1,\ldots,s$ are introduced such that $\xi_k = 1$ corresponds to a violated constraints (i.e. to a negative pattern covered by the implicant). For this reason the sum $\sum_{k=s_1+1}^{s} \xi$, which is just the number of misclassified patterns, must be less than $\varepsilon_0 s_0$. If the training set is noisy, the optimal implicant can be found by solving the following LP problem, where it is supposed that the first s'_1 positive patterns are not yet covered:

$$\begin{aligned}
&\min_{\xi,\mathbf{a}} \ \frac{\nu}{s'_1}\sum_{k=1}^{s'_1}\xi_k + \frac{\lambda}{s_1 - s'_1}\sum_{k=s'_1+1}^{s_1}\xi_k + \frac{\mu}{m}\sum_{i=1}^{m} a_i + \frac{\omega}{s_0}\sum_{k=s_1+1}^{s}\xi_k \\
&\text{subj to } \sum_{i=1}^{m} a_i(1-z_{ki}) = \xi_k \quad \text{for } k = 1,\ldots,s_1 \\
&\sum_{i=1}^{m} a_i(1-z_{ki}) \geq 1-\xi_k \quad \text{for } k = s_1+1,\ldots,s \qquad (6)\\
&\sum_{k=s_1+1}^{s}\xi_k \leq \varepsilon_0 s_0\,, \quad a_i \in \{0,1\} \quad \text{for } i = 1,\ldots,m \\
&\xi_k \geq 0 \quad \text{for } k = 1,\ldots,s_1\,, \quad \xi_k \in \{0,1\} \quad \text{for } k = s_1+1,\ldots,s
\end{aligned}$$

If desired, only the implicants covering at least a fraction η_1 of positive examples may be generated. To this aim it is sufficient to add the following constraint

$$\sum_{k=1}^{s_1}\xi_k \leq (1-\eta_1)s_1$$

Notice that further requirements have to be imposed when dealing with real-world problems. In fact, due to the coding (2) or (3) adopted in the latticization phase, only some implicants correspond to a condition consistent with the original inputs. In particular at least one zero must be present in the substring relative to each input variable.

5.2 Implicant Selection

Once the antichain A' has been generated, it will be useful to look for a subset A of A' which is able to describe the data in the training set with sufficient accuracy. To this aim both the number of implicants included in A and the number N_k of nonnull components in each element $\mathbf{a}_k \in A$ must be minimized.

Denote with $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_q$ the q implicants obtained in the generation step and with c_{kj} a binary variable asserting if the input vector $\mathbf{z}_k$, $k = 1, \ldots, s_1$, is covered by $\mathbf{a}_j$:

$$c_{kj} = \begin{cases} 1 & \text{if } \mathbf{z}_k \text{ is covered by } \mathbf{a}_j \\ 0 & \text{otherwise} \end{cases}$$

In addition, consider the binary vector ζ having as jth component the value $\zeta_j = 1$ if the corresponding implicant $\mathbf{a}_j$ is included in the final collection A. Then, an optimal subset $A \subset A'$ can be found by solving the following constrained optimization problem:

$$\begin{aligned} &\min_{\zeta} \sum_{j=1}^{q} \zeta_j (\alpha + \beta N_j)) \\ &\text{subj to } \sum_{j=1}^{q} c_{kj} \zeta_j \geq 1 \quad \text{for } k = 1, \ldots, s_1 \\ &\zeta_j \in \{0,1\} \quad \text{for } j = 1, \ldots, q \end{aligned} \tag{7}$$

where α and β are constants.

Additional requirements can be added to the problem (7) in order to improve the generalization ability of A. For example, if the presence of noise has to be taken into account, the antichain A can be allowed not to cover a small fraction of positive examples. In this case, the problem (7) becomes

$$\begin{aligned} &\min_{\zeta} \sum_{j=1}^{q} \zeta_j (\alpha + \beta N_j)) \\ &\text{subj to } \sum_{j=1}^{q} c_{kj} \zeta_j \geq 1 - \xi_k \quad \text{for } k = 1, \ldots, s_1 \\ &\sum_{j=1}^{q} c_{kj} \zeta_j \leq \xi_k \quad \text{for } k = s_1, \ldots, s \\ &\sum_{k=1}^{s_1} \xi_k \leq \varepsilon s_1 \qquad \sum_{k=s_1+1}^{s} \xi_k \leq \varepsilon s_0 \qquad \zeta_j \in \{0,1\} \quad \text{for } j = 1, \ldots, q \\ &\xi_j \in \{0,1\} \quad \text{for } j = 1, \ldots, s_1 \qquad \xi_j \geq 0 \quad \text{for } j = s_1, \ldots, s \end{aligned} \tag{8}$$

5.3 An Approximate Method for Solving the LP Problem

The solution of the problems (4) (or (6)) and (7) (or (8)) allows the generation of a minimal set of implicants for the problem at hand. Nevertheless, in the presence of a large amount of data, the number of variables and constraints for the LP problems increases considerably, thus making the SP algorithm very slow.

Conversion of the continuous solution into a binary vector

1 Set $a_i = 0$ for each $i = 1, \ldots, m$.

2 While the constraints in (4) are violated

a. Select $\hat{\imath} = \arg\max_i a_i$.
b. Set $a_{\hat{\imath}} = 1$.

3. While the constraints in (4) are satisfied

a. Pick an index $\hat{\imath}$.
b. Set $a_{\hat{\imath}} = 0$.
c. If a constraints in (4) is violated, set $a_{\hat{\imath}} = 1$.

Fig. 3 A greedy method for converting a continuous solution of problem (4) into a binary vector.

In this subsection, an approximate algorithm able to reduce the execution time of the training algorithm for huge datasets will be introduced. The method will be described for the minimization problem (4), as its generalization to the case of (6), (7) or (8) is straightforward.

Most of the LP methods perform the minimization of a function $\Phi(\mathbf{a})$ through the following phases:

1. A continuous solution $\mathbf{a}_c$ is retrieved by suppressing the constraints $a_i \in \{0, 1\}$.
2. Starting from $\mathbf{a}_c$ the optimal binary vector $\mathbf{a}$ is obtained through a branch and bound approach.

In particular, during phase 2, the algorithm must ensure that all the constraints of the original LP problem (4) are still satisfied by the binary solution. The search for an optimal integer solution can thus require the exploration of many combinations of input values. Preliminary tests have shown that, when the number of integer input variables in an LP problem increases, the time employed by Phase 2 may be much longer than that needed by Phase 1.

For this reason, an approximate algorithm will be proposed to reduce the number of combinations explored in the conversion of the continuous solution to the binary one.

Of course, the higher the number of 1s in a vector $\mathbf{a}$, the higher is the probability that it satisfies all the constraints in (4), since the number of covered patterns is usually smaller. Therefore, the vector $\mathbf{a}_c$ can be employed in order to retrieve a minimal subset of indexes to be set to one, starting from the assumption that a higher value of $(a_c)_i$ corresponds to a higher probability that a_i has to be set to 1 (and vice versa).

The algorithm for the conversion of the continuous solution to a binary one is shown in Fig. 3. The method starts by setting $a_i = 0$ for each i; then the a_i corresponding to the highest value of $(a_c)_i$ is set to 1. The procedure is repeated controlling at each iteration if the constraints (4) are satisfied. When no constraint is violated the procedure is stopped; smart lattice descent techniques [14] may be adopted to further reduce the number of active bits in the implicant.

These methods are based on the definition of proper criteria in the choice of the bit to be set to zero. Of course, when the implicant does not satisfy all the constraints, the bit is set to one again, the algorithm is stopped and the resulting implicant is added to the antichain A. The same approach may be employed in the the pruning phase, too.

The approximate version of the SP algorithm, obtained by employing the greedy procedure for transforming the continuous solution of each LP problem involved in the method into a binary one, is named *Approximate Switch Programming* (*ASP*).

6 Transforming the Implicants into a Set of Rules

If the discretization task described in Subsection 4.1 is employed to construct the latticizers, every implicant $\mathbf{a} \in \{0,1\}^m$ generated by SC can be translated into an intelligible rule underlying the classification at hand. This assertion can be verified by considering the substrings $\mathbf{a}_i$ of $\mathbf{a}$ that are associated with the ith input x_i to the network. The logical product $\bigwedge_{j \in P(\mathbf{a})} z_j$, performed by the AND port corresponding to $\mathbf{a}$, gives output 1 only if the binary string $\mathbf{z} = \varphi(\mathbf{x})$ presents a value 1 in all the positions where $\mathbf{a}_i$ has value 1.

If x_i is an ordered variable, this observation gives rise to the condition $x_i \in \bigcup_{j \in I_{m_i} \setminus P(\mathbf{a}_i)} R_{ij}$. However, in the analysis of real-world problems, the execution of SP and ASP is constrained to generate only binary strings $\mathbf{a}_i$ (for ordered variables) having a single sequence of consecutive values 0, often called a *run* of 0. In this case the above condition can simply be written in one of the following three ways:

- $x_i \leq r_{ij}$, if the run of 0 begins at the first position and finishes at the jth bit of $\mathbf{a}_i$,
- $r_{ij} < x_i \leq r_{ik}$, if the run of 0 begins at the $(j+1)$th position and finishes at the kth bit of $\mathbf{a}_i$,
- $x_i > r_{ij}$, if the run of 0 begins at the $(j+1)$th position and finishes at the last (m_ith) bit of $\mathbf{a}_i$.

As an example, suppose that an ordered variable x_i has been discretized by using the four cutoffs $0.1, 0.25, 0.3, 0.5$. If the implicant $\mathbf{a}$ with $\mathbf{a}_i = 10011$ has been produced by SC, the condition $0.1 < x_i \leq 0.3$ has to be included in the **if** part of the **if-then** rule associated with $\mathbf{a}$.

On the other hand, if x_i is a nominal variable the portion $\mathbf{a}_i$ of an implicant $\mathbf{a}$ gives rise to the condition $x_i \in \bigcup_{j \in I_{m_i} \setminus P(\mathbf{a}_i)} \{j\}$. Again, if the implicant $\mathbf{a}$ with $\mathbf{a}_i = 01101$ has been produced by SC, the condition $x_i \in \{1,4\}$ has to be included in the **if** part of the **if-then** rule associated with $\mathbf{a}$. In any case, if the binary string $\mathbf{a}_i$ contains only values 0, the input x_i will not be considered in the rule for $\mathbf{a}$.

Thus, it follows that every implicant **a** gives rise to an **if-then** rule, having in its **if** part a conjunction of the conditions obtained from the substrings $\mathbf{a}_i$ associated with the d inputs x_i. If all these conditions are verified, to the output $y = \hat{g}(\mathbf{x})$ will be assigned the value 1.

Due to this property, SP and ASP (with the addition of discretization and latticization) become rule generation methods, being capable of retrieving from the training set some kind of intelligible information about the physical system underlying the binary classification problem at hand.

7 An Example of SNN Training

A simple example will be presented in this section in order to make the training of an SNN clearer. Consider the problem of forecasting the quality of the layer produced by a rolling mill starting from the knowledge of two continuous values: Pressure (x_1) and rolling Speed (x_2). The behavior of the rolling mill can be described by a function $g : \mathbb{R}^2 \to \{0,1\}$, whose output y may be either 0 (Bad layer) or 1 (Good layer). The aim of the classification task is therefore to realize a function $\hat{g}$ which constitutes a valid approximation for g starting from the training set S shown in Tab. 1.

As Tab. 1 shows, S is composed of 20 examples: 10 of those are Good and 10 are Bad. Suppose that the discretization process has subdivided the values of Pressure into three intervals $(-\infty, 1.63)$, $(1.63, 1.56)$, $(2.56, \infty)$, whereas the domain for Speed has been partitioned into 4 intervals $(-\infty, 3.27)$, $(3.27, 4.9)$, $(4.9, 6.05)$, $(6.05, \infty)$.

Through the discretization phase, it is possible to define a mapping $\psi : \mathbb{R}^2 \to I_3 \times I_4$, which associates two integer values v_1 and v_2 with each input pattern $\mathbf{x}$. Starting from the vector $\mathbf{v}$ obtained at the discretization step, the latticizer associates a binary string $\mathbf{z} = \varphi(\mathbf{x})$ with each pattern $\mathbf{x}$. Since the number of intervals for x_1 and x_2 is respectively 3 and 4, the latticizer produces a 7-dimensional binary string, obtained by concatenating the substrings relative to each input variable.

For example, the input pattern $\mathbf{x} = (3.12, 3.90)$ belongs to the interval $(2.56, \infty)$ for x_1 and to the interval $(3.27, 4.90)$ for x_2. Therefore the integer vector $\mathbf{v} = (3, 2)$ and the binary string 1101011, obtained through the inverse-only one coding, are associated with $\mathbf{x}$. Starting from the original data, it is possible to obtain a binary training set S', which is used to perform the classification. In fact the function $\hat{g}$ can be retrieved as $\hat{g}(\mathbf{x}) = \sigma(F(\varphi(\mathbf{x})))$.

Since the expected noise in the training data is negligible, the problem (6) can be solved by setting $\varepsilon = \omega = 0$, whereas a standard choice for the other coefficients in the cost function is given by $\lambda = \mu = \nu = 1$.

Suppose that the neurons for the output class 1 are generated initially. A first execution of the SP algorithm on the rolling mill problem produces the implicant 1000001. In fact it can be easily verified that this binary string satisfies all the constraints in (6) since it does not cover any example labelled by 0. Moreover, it covers 7 positive examples (all those after the third one) and has only two active bits, thus

Table 1 The original and transformed dataset for the problem of controlling the quality of a layer produced by a rolling mill. x_1 and x_2 are the original values for Pressure and Speed, v_1 and v_2 are the discretized values, whereas $\mathbf{z}$ is the binary string obtained through the latticizer. The quality of the resulting layer is specified by the value of the Boolean variable y.

x_1	x_2	v_1	v_2	$\mathbf{z}$	y
0.62	0.65	1	1	0110111	1
1.00	1.95	1	1	0110111	1
1.31	2.47	1	1	0110111	1
1.75	1.82	2	1	1010111	1
2.06	3.90	2	2	1011011	1
2.50	4.94	2	3	1011101	1
2.62	2.34	3	1	1100111	1
2.75	1.04	3	1	1100111	1
3.12	3.90	3	2	1101011	1
3.50	4.94	3	3	1011110	1
0.25	5.20	1	3	0111101	0
0.87	6.01	1	3	0111101	0
0.94	4.87	1	2	0111011	0
1.87	4.06	1	2	0111011	0
1.25	8.12	1	4	0111110	0
1.56	6.82	1	4	0111110	0
1.87	8.75	2	4	1011110	0
2.25	8.12	2	4	1011110	0
2.50	7.15	2	4	1011110	0
2.81	9.42	3	4	1101110	0

scoring a very low value of the objective function. Nevertheless, the first three examples are still to be covered, so the SP algorithm must be iterated.

The constraint (5) has to be added

$$(1-a_1)+(1-a_7)\geq 1$$

and the first three examples constituting the set S'' must be considered separately in the cost function (6).

A second execution of the SP algorithm generates the implicant 0000111, which covers 6 examples among which are the ones not yet covered. Therefore the antichain $A=\{1000001,0000111\}$, corresponding to the PDNF $f(\mathbf{z})=z_1z_7+z_5z_6z_7$, correctly describes all the positive examples. It is also minimal since the pruning phase cannot eliminate any implicant.

In a similar way an antichain is generated for the output class labelled by 0, thus producing the second layer of the SNN.

A possible choice for the weight u_h to be associated with the h-th neuron is given by its *covering*, i.e. the fraction of examples covered by it. For example, the weights associated with the neurons for the class 1 may be $u_1=0.7$ for 1000001 and $u_2=0.6$ for 0000111.

In addition the retrieved implicants can be transformed in intelligible rules involving the input variables. For example the implicant 1000001, associated with the output class 1, corresponds to the rule:

$$\textbf{if} \quad x_1 < 1.63 \ \text{AND} \ x_2 > 6.05 \quad \textbf{then} \quad y = 1$$

8 Simulation Results

To obtain a preliminary evaluation of the performances achieved by SNNs trained with SP or ASP, the classification problems included in the well-known StatLog benchmark [9] have been considered. In this way the generalization ability and the complexity of resulting SNNs can be compared with those of other machine learning methods, among which are the backpropagation algorithm (BP) and rule generation techniques based on decision trees, such as C4.5 [15].

All the experiments have been carried out on a personal computer with an Intel Core Quad Q6600 (CPU 2.40 GHz, RAM 3 GB) running under the Windows XP operative system.

The tests contained in the Statlog benchmark presents different characteristics which allow the evaluation of different peculiarities of the proposed methods. In particular, four problems (Heart, Australian, Diabetes, German) have a binary output; two of them (Heart and German) are clinical datasets presenting a specific weight matrix which aims to reduce the number of misclassifications on ill patients. The remaining datasets present 3 (Dna), 4 (Vehicle) or 7 (Segment, Satimage, Shuttle) output classes. In some experiments, the results are obtained through a cross-validation test; however, in the presence of large amount of data, a single trial is performed since the time for many executions may be excessive.

The generalization ability of each technique is evaluated through the level of misclassification on a set of examples not belonging to the training set; on the other hand, the complexity of an SNN is measured using the number of AND ports in the second layer (corresponding to the number of intelligible rules) and the average number of conditions in the **if** part of a rule. Tab. 2 presents the results obtained on the datasets, reported in increasing order of complexity. Accuracy and complexity of resulting SNNs are compared to those of rulesets produced by C4.5. In the same table is also shown the best generalization error included in the StatLog report [9] for each problem, together with the rank scored by SNN when its generalization error is inserted into the list of available results.

The performances of the different techniques for training an SNN depend on the characteristics of the different problems. In particular the SP algorithm scores a better level of accuracy with respect to ASP in the datasets Heart and Australian. In fact, these problems are characterized by a small amount of data so that the execution of the optimal minimization algorithm may obtain a good set of rules within a reasonable execution time.

On the other hand, the misclassification of ASP is lower than that of SP in all the other problems (except for Shuttle), which are composed of a greater amount of

Table 2 Generalization error of SNN, compared with C4.5, BP and other methods, on the StatLog benchmark.

Test Problem	Generalization error					Rank	
	SP	ASP	C4.5	BP	Best	SP	ASP
HEART	0.439	0.462	0.781	0.574	0.374	6	9
AUSTRALIAN	0.138	0.141	0.155	0.154	0.131	3	3
DIABETES	0.246	0.241	0.270	0.248	0.223	7	5
VEHICLE	0.299	0.291	0.266	0.207	0.150	18	15
GERMAN	0.696	0.568	0.985	0.772	0.535	10	3
SEGMENT	0.0424	0.042	0.040	0.054	0.030	8	8
DNA	0.0658	0.056	0.076	0.088	0.041	7	3
SATIMAGE	0.168	0.149	0.150	0.139	0.094	19	10
SHUTTLE	0.0001	0.0001	0.001	0.43	0.0001	1	1

data. The decrease of the performances of SP in the presence of huge datasets is due to the fact that simplifications in the LP problem are necessary in order to make it solvable within a reasonable period of time. For example, some problems may be solved by setting $\varepsilon = \omega = 0$, since taking into account the possible presence of noise gives rise to an excessive number of constraints in (6).

Notice that in one case (Shuttle), SP and ASP achieve the best results among the methods in the StatLog archive, whereas in four other problems ASP achieves one of the first five positions. However, ASP is in the first ten positions in all the problems except for Vehicle.

Moreover, a comparison of the other methods reported in Tab. 2 with the best version of SNN for each problem illustrates that:

- Only in one case (Vehicle) the classification accuracy achieved by C4.5 is higher than that of SNN; in two problems (Satimage and Segment) the performances are similar, whereas in all the other datasets SNN scores significantly better results.
- In two cases (Vehicle and Satimage), BP achieves better results with respect to SNN; in all the other problems the performances of SNN are significantly better than those of BP.

These considerations highlight the good quality of the solutions offered by the SNNs, trained by the SP or ASP algorithm.

Nevertheless, the performances obtained by SP are conditioned by the number of examples s in the training set and by the number of input variables d. Since the number of constrains in (4) or (6) depends linearly on s, SP becomes slower and less efficient when dealing with complex training sets. In particular, the number of implicants generated by SP in many cases is higher than that of the rules obtained by C4.5, causing an increase in the training time.

However a smart combination of the standard optimization techniques with the greedy algorithm in Sec. 5.3 may allow complex datasets to be handled very efficiently. In fact the execution of ASP requires at most three minutes for each

execution of the first six problems, about twenty minutes for the Dna dataset and about two hours for Satimage and Shuttle.

Notice that the minimization of (4) or (6) is obtained using the package *Gnu Linear Programming Kit* (*GLPK*) [8], a free library for the solution of linear programming problems. It is thus possible to improve the above results by adopting more efficient tools to solve the LP problem for the generation of implicants.

Concluding Remarks

In this paper a general schema for constructive methods has been presented and employed to train a Switching Neural Network (SNN), a novel connectionist model for the solution of classification problems. According to the SNN approach, the input-output pairs included in the training set are mapped to Boolean strings according to a proper transformation which preserves ordering and distance. These new binary examples can be viewed as a portion of the truth table of a positive Boolean function f, which can be reconstructed using a suitable algorithm for logic synthesis.

To this aim a specific method, named Switch Programming (SP), for reconstructing positive Boolean functions from examples has been presented. SP is based on the definition of a proper integer linear programming problem, which can be solved with standard optimization techniques. However, since the treatment of complex training sets with SP may require an excessive computational cost, a greedy version, named Approximate Swith Programming (ASP), has been proposed to reduce the execution time needed for training SNN.

The algorithms SP and ASP have been tested by analyzing the quality of the SNNs produced when solving the classification problems included in the Statlog archive. The results obtained show the good accuracy of classifiers trained with SP and ASP. In particular, ASP turns out to be very convenient from a computational point of view.

Acknowledgement. This work was partially supported by the Italian MIUR project "Laboratory of Interdisciplinary Technologies in Bioinformatics (LITBIO)".

References

1. Boros, E., Hammer, P.L., Ibaraki, T., Kogan, A., Mayoraz, E., Muchnik, I.: An implementation of Logical Analysis of Data. IEEE Trans. Knowledge and Data Eng. 9, 292–306 (2004)
2. Devroye, L., Györfi, L., Lugosi, G.: A probabilistic theory of Pattern Recognition. Springer, Heidelberg
3. Ferrari, E., Muselli, M.: A constructive technique based on linear programming for training switching neural networks. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part II. LNCS, vol. 5164, pp. 744–753. Springer, Heidelberg (2008)
4. Hong, S.J.: R-MINI: An iterative approach for generating minimal rules from examples. IEEE Trans. Knowledge and Data Eng. 9, 709–717 (1997)

5. Kerber, R.: ChiMerge: Discretization of numeric attributes. In: Proc. 9th Int'l. Conf. on Art. Intell., pp. 123–128 (1992)
6. Kohavi, R., Sahami, M.: Error-based and Entropy based discretization of continuous features. In: Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining, pp. 114–119 (1996)
7. Liu, H., Setiono, R.: Feature selection via discretization. IEEE Trans. on Knowledge and Data Eng. 9, 642–645 (1997)
8. Makhorin, A.: GNU Linear Programming Kit - Reference Manual (2008), `http://www.gnu.org/software/glpk/`
9. Michie, D., Spiegelhalter, D., Taylor, C.: Machine Learning, Neural and Statistical Classification. Ellis-Horwood, London (1994)
10. Muselli, M.: Sequential Constructive Techniques. In: Leondes, C. (ed.) Optimization Techniques, Neural Netw. Systems Techniques and Applications, pp. 81–144. Academic Press, San Diego (1998)
11. Muselli, M.: Approximation properties of positive boolean functions. In: Apolloni, B., Marinaro, M., Nicosia, G., Tagliaferri, R. (eds.) WIRN 2005 and NAIS 2005. LNCS, vol. 3931, pp. 18–22. Springer, Heidelberg (2006)
12. Muselli, M.: Switching neural networks: A new connectionist model for classification. In: Apolloni, B., Marinaro, M., Nicosia, G., Tagliaferri, R. (eds.) WIRN/NAIS 2005. LNCS, vol. 3931, pp. 23–30. Springer, Heidelberg (2006)
13. Muselli, M., Liberati, D.: Binary rule generation via Hamming Clustering. IEEE Trans. Knowledge and Data Eng. 14, 1258–1268 (2002)
14. Muselli, M., Quarati, A.: Reconstructing positive Boolean functions with Shadow Clustering. In: Proc. 17th European Conf. Circuit Theory and Design, pp. 377–380 (2005)
15. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1994)
16. Reed, R.: Pruning Algorithm–A Survey. IEEE Trans. on Neural Netw. 4, 740–747 (1993)
17. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by backpropagating errors. Nature 323, 533–536 (1988)
18. Vapnik, V.N.: Statistical learning theory. John Wiley & Sons, New York (1998)