A Digital Architecture for Support Vector Machines: Theory, Algorithm, and FPGA Implementation

Davide Anguita, *Member, IEEE*, Andrea Boni, and Sandro Ridella, *Member, IEEE*

***Abstract*—In this paper, we propose a digital architecture for support vector machine (SVM) learning and discuss its implementation on a field programmable gate array (FPGA). We analyze briefly the quantization effects on the performance of the SVM in classification problems to show its robustness, in the feedforward phase, respect to fixed-point math implementations; then, we address the problem of SVM learning. The architecture described here makes use of a new algorithm for SVM learning which is less sensitive to quantization errors respect to the solution appeared so far in the literature. The algorithm is composed of two parts: the first one exploits a recurrent network for finding the parameters of the SVM; the second one uses a bisection process for computing the threshold. The architecture implementing the algorithm is described in detail and mapped on a real current-generation FPGA (Xilinx Virtex II). Its effectiveness is then tested on a channel equalization problem, where real-time performances are of paramount importance.**

***Index Terms*—Digital neuroprocessors, field programmable gate arrays (FPGAs), quantization effects, support vector machine (SVM).**

I. INTRODUCTION

THE hardware implementation of neural networks has recently attracted new interest from the neurocomputing community despite the skepticism generated by the devices of the first generation, which appeared during the 1980s (see, for example, [1] for a survey of these solutions). Nowadays, two research areas can be easily identified: the first one focuses on biological inspired devices and builds on Mead's seminal work [2]. In this case, the physical behavior of microelectronic devices is exploited to realize complex information processing with very low requirements in terms of area and power consumption [3].

The second area addresses the hardware implementation of algorithms, which are inspired by the neurocomputing framework but not necessarily justified form a biological point of view. Its main target is the design of dedicated analog or digital hardware with improved characteristics (e.g., performance, silicon area, power consumption, etc.) respect to a software implementation on a general-purpose microprocessor. This approach has been the most criticized in the past [4], mainly because one of the main targets of the research has been the raw computational power. There are, however, many other advantages in designing special-purpose devices, as recalled before, and in some cases (e.g., embedded systems) a dedicated solution can be preferable [5].

Our work fits in this last framework: we propose a new algorithm for support vector machine (SVM) learning and a digital architecture that implements it. The SVM is a new learning-by–example paradigm recently proposed by Vapnik and based on its statistical learning theory [6]. After the first preliminary studies, SVMs have shown a remarkable efficiency, especially when compared with traditional artificial neural networks (ANNs), like the multilayer perceptron. The main advantage of SVM, with respect to ANNs, consists in the structure of the learning algorithm, characterized by the resolution of a constrained quadratic programming problem (CQP), where the drawback of local minima is completely avoided. Our algorithm improves on the proposals appeared so far (e.g., [17] and consists of two parts: the first one, previously reported in the literature, solves the CQP respect all the parameters of the network, except the threshold, while the second one allows the computation of such threshold by using an iterative procedure.

The proposed algorithm can be easily mapped to a digital architecture: to assess the effectiveness of our approach, and measure the actual performance of both the algorithm and the architecture, we implement our solution on a field programmable gate array (FPGA) and test it on a telecommunication application. Our choice is motivated by recent advances in the FPGA-based technology, which allows easy reprogrammability, fast development times and reduced efforts with respect to full-custom very large-scale integration (VLSI) design. At the same time, the advances in the microelectronics process technology allow the design of FPGA-based digital systems having performances very close to the ones obtained by a manual full-custom layout (see, for example, [25], which details an efficient implementation of a neural processor).

In Section II, we revise briefly the SVM. Section III addresses the hardware implementation of SVMs when targeting digital solutions. New results on the robustness of this learning machine, respect to quantization errors, are presented, and, after surveying the state of the art of SVM learning algorithms, targeted to VLSI implementations, our new proposal is detailed. The digital architecture is described in Section IV and its FPGA implementation, along with the experiments on a telecommunication application, is described in Section V.

II. SVM

The objective of SVM learning is finding a classification function $\hat{\Phi}\colon X \to Y$ that approximates the unknown Φ, on the

Manuscript received September 15, 2002.

D. Anguita and S. Ridella are with the Department of Biophysical and Electronic Engineering, University of Genova, 16145 Genoa, Italy (e-mail: anguita@dibe.unige.it; ridella@dibe.unige.it)

A. Boni is with the Department of Information and Communication Technologies, University of Trento, Trento, Italy (e-mail: andrea.boni@ing.unitn.it)

Digital Object Identifier 10.1109/TNN.2003.816033

basis of the set of measures $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$, where $\boldsymbol{x}_i \in X \subseteq \Re^l$ is an input pattern, and $y_i \in Y = \{-1, +1\}$ the corresponding target.

To allow for nonlinear classification functions, the training points are mapped from the input space X to a *feature space* $Z \subseteq \Re^L$, with $L \gg l$, through a nonlinear mapping $\varphi: X \to Z$; then, a simple linear hyperplane is used for separating the points in the feature space. By using clever mathematical properties of nonlinear mappings, SVMs avoid explicitly working in the feature space, so that the advantages of the linear approach are retained even though a nonlinear separating function is found (see [6]–[8] for more details)

The function $\hat{\Phi}$ is given by

$$\hat{\Phi}(\boldsymbol{x}) = \boldsymbol{w} \cdot \varphi(\boldsymbol{x}) + b \tag{1}$$

where $\boldsymbol{w}$ is the normal vector of the separating hyperplane.

Among all possible hyperplanes, SVMs find the one that corresponds to a function $\hat{\Phi}$ having a maximal margin or, in other words, the maximum distance from the points of each class. Since it is not common that all the points can be correctly classified, even if mapped in the high-dimensional feature space Z, the SVM allows for some errors but penalizes their cardinality.

Formally, this description leads to a CQP, by requiring us to find

$$\min_{\boldsymbol{w},b}\left[\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^m \xi_i\right] \tag{2}$$

subject to

$$y_i(\boldsymbol{w} \cdot \varphi(\boldsymbol{x}_i) + b) \geq 1 - \xi_i, \quad \forall\, i = 1, \ldots, m. \tag{3}$$

This is usually referred as the *Primal* $(\mathcal{P})$ CQP. The function to be minimized is composed by two parts: the first one forces the hyperplane to have a maximal margin, while the second term penalizes the presence of misclassified points. The constant C simply sets the tradeoff between the two terms.

One of the reason of the SVM success is the fact that this approach can be used to control the complexity of the learning machine, namely through the structural risk minimization principle [6]. This principle can provide a method for controlling the complexity of the learning machine and define upper bounds of its generalization ability, albeit in a statistical framework.

The above $\mathcal{P}$—CQP is usually rewritten in dual form $(\mathcal{D})$, by using the Lagrange multiplier theory [6]

$$\min_\alpha \frac{1}{2}\sum_{i,j=1}^m \alpha_i\alpha_j y_i y_j \varphi(\boldsymbol{x}_i)\cdot\varphi(\boldsymbol{x}_j) - \sum_{i=1}^m \alpha_i \tag{4}$$

subject to

$$\begin{aligned} 0 \leq &\alpha_i \leq C \\ \sum_{i=1}^m \alpha_i y_i &= 0 \end{aligned} \tag{5}$$

where

$$\boldsymbol{w} = \sum_{i=1}^m \alpha_i y_i \varphi(\boldsymbol{x}) \tag{6}$$

TABLE I
ALGORITHM 1: DSVM WITH FIXED BIAS

```
Step  Description
1.    input: $Q$, $b$, $\boldsymbol{\alpha}^0$;
2.    set $\tilde{\boldsymbol{r}} = (\boldsymbol{r} - \boldsymbol{y}b)$;   $\eta = \frac{2}{m \max_{i,j}(q_{ij})}$
3.    set endrun=false;   $k = 0$
4.    while not endrun do
5.        $\boldsymbol{g} = \left(-Q\boldsymbol{\alpha}^k + \tilde{\boldsymbol{r}}\right)$
6.        $\boldsymbol{z}^{k+1} = \boldsymbol{\alpha}^k + \eta\boldsymbol{g}$
7.        $\alpha_i^{k+1} = \max\left(0, \min\left(z_i^{k+1}, C\right)\right), \forall i = 1, \ldots, m$
8.        if $\forall i \left(\alpha_i^{k+1} - \alpha_i^k\right) \leq \varepsilon$ then
8.            endrun=true
10.       else k=k+1
11.   enddo
12.   output: $\boldsymbol{\alpha}' = \boldsymbol{\alpha}^{k+1}$
```

which allows us to write $\hat{\Phi}$ using the dual variables

$$\hat{\Phi}(\boldsymbol{x}) = \sum_{i=1}^m \alpha_i y_i \boldsymbol{\varphi}(\boldsymbol{x}_i)\cdot\boldsymbol{\varphi}(\boldsymbol{x}) + b. \tag{7}$$

There are several advantages in using the dual formulation: the main one is that there is no need to know explicitly the function $\boldsymbol{\varphi}$, but only the inner product between two points in the feature space. This is the well-known *kernel trick* that allows to deal implicitly with nonlinear mappings through the use of kernel functions

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\varphi}(\boldsymbol{x}_i)\cdot\boldsymbol{\varphi}(\boldsymbol{x}_j). \tag{8}$$

Using the above notation, the $\mathcal{D}$-CQP can be rewritten in a compact form, as follows:

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \left[\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \boldsymbol{r}^T\boldsymbol{\alpha}\right] \\ &0 \leq \boldsymbol{\alpha} \leq C \\ &\boldsymbol{\alpha}^T\boldsymbol{y} = 0 \end{aligned} \tag{9}$$

where $q_{ij} = y_i y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $\boldsymbol{r}$ is a vector of all ones.

Since the seminal works on kernel functions, many kernels of the form given by (8) have been found; among them are the linear, the Gaussian, and the polynomial kernels

$$\begin{aligned} k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= \boldsymbol{x}_i\cdot\boldsymbol{x}_j \\ k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= e^{(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/2\sigma^2)} \\ k(\boldsymbol{x}_i, \boldsymbol{x}_j) &= (1 + \boldsymbol{x}_i\cdot\boldsymbol{x}_j)^p. \end{aligned} \tag{10}$$

Another advantage of using the kernel functions lies in the fact that they are positive semidefinite functionals. Therefore,

TABLE II
ALGORITHM 2: FIBS

Step	Description
1.	$b_{low} = b = -1$; $\boldsymbol{\alpha}^0 = 0$; $b_{up} = +1$; endFibs=false
2.	$\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \text{sgn}\left(y^T\boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$
3.	if $s < 0$ then
4.	while $s < 0$ do
5.	$b_{up} = b$; $b = 2b$; $b_{low} = 2b_{low}$
6.	$\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \text{sgn}\left(y^T\boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$
7.	enddo
8.	else if $s > 0$ then
9.	$b = b_{up}$
10.	$\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \text{sgn}\left(y^T\boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$
11.	if $s > 0$ then
12.	while $s > 0$ do
13.	$b_{low} = b$; $b = 2b$; $b_{up} = 2b_{up}$
14.	$\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \text{sgn}\left(y^T\boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$
15.	enddo
16.	else endFibs = true
17.	while not endFibs do
18.	$b = (b_{low} + b_{up})/2$
19.	$\boldsymbol{\alpha}' = \mathcal{A}_b(Q, b, \boldsymbol{\alpha}^0)$; $s = \text{sgn}\left(y^T\boldsymbol{\alpha}'\right)$; $\boldsymbol{\alpha}^0 = \boldsymbol{\alpha}'$
20.	if $s = 0$ or $(b_{low} - b_{up}) \leq \varepsilon_b$ then endFibs=true
21.	else if $s < 0$ then $b_{up} = b$
22.	else $b_{low} = b$
23.	enddo

using this property and the fact that the constraints of the above optimization problem are affine, any local minima is also a global one and algorithms exist which, given a fixed tolerance, find the solution in a finite number of steps [9]. Furthermore, if the kernel is strictly positive definite, which is always the case except for pathological situations, the solution is also unique. These properties overcome many typical drawbacks of traditional neural-network approaches, such as the determination of a suitable minimum, the choice of the starting point, the optimal stopping criteria, etc.

As a final remark, note that the threshold b does not appear in the dual formulation, but it can be found by using the Karush–Khun–Tucker (KKT) conditions at optimality. Let $SV = \{i : \alpha_i \in (0, C)\}$ be the set of *true support vectors,* then the following equality holds:

$$\sum_{j=1}^{m} \alpha_j y_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) + b = y_i \tag{11}$$

thus, b can be found by

$$b = \frac{1}{|SV|} \sum_{i \in SV} \left(y_i - \sum_{j=1}^{m} \alpha_j y_j k(x_i, x_j) \right). \tag{12}$$

Actually, this approach is correct as long as $SV \neq \emptyset$; otherwise one can use a more robust method, suggested in [10] (see also [9]), where the threshold is computed using the KKT conditions, but without resorting to support vectors.

III. Hardware Implementation of SVMs

A. New VLSI-Friendly Algorithm for SVM Learning

Optimization problems like $\mathcal{D}$—CQP are well known to the scientific community, as usually faced when solving several real-world tasks. As a consequence, they have been deeply studied by researchers, and several methods have been proposed for their resolution. Among others, methods that can be easily implemented in hardware are particularly appealing: we refer to them as VLSI-friendly algorithms. The leading idea of these methods is to map the problem on a dynamical system described by a differential equation

$$\dot{\boldsymbol{\nu}} = F_A(\boldsymbol{\nu}) \tag{13}$$

with $\boldsymbol{\nu}^T = \left[\boldsymbol{\alpha}^T, b\right]$ and whose stable point, for $t \to \infty$, coincides with the solution of the optimization problem.

Equation (13) can be seen as a recurrent neural network and, from an electronic point of view, can be implemented, on analog hardware, with simple electronic devices [14], [15].

A digital architecture can be targeted in a similar way by defining a recurrent relation of the form

$$\boldsymbol{\nu}^{k+1} = F_D\left(\boldsymbol{\nu}^k\right). \tag{14}$$

A simple way to obtain (14) from (13) is to use the Euler's method for integration, and to obtain

$$\boldsymbol{\nu}^{k+1} = \boldsymbol{\nu}^k + \eta F_A\left(\boldsymbol{\nu}^k\right) \tag{15}$$

where η is the integration step. Unfortunately, finding a suitable η, which guarantees the convergence of $\boldsymbol{\nu}^k$ toward the solution when $k \to \infty$, is not a trivial task.

Recently, a useful convergence result has been presented for a network that solves a CQP with inequality constraints [16]. As shown in [17], this algorithm (digital SVM or DSVM) can be applied effectively to SVM learning when a Gaussian kernel is chosen. The underlying idea is to exploit the fact that a Gaussian kernel maps the data to an infinite feature space, so the effect of removing one of the parameters of the SVM from the learning process can be negligible. In particular, if we let $b = 0$, we

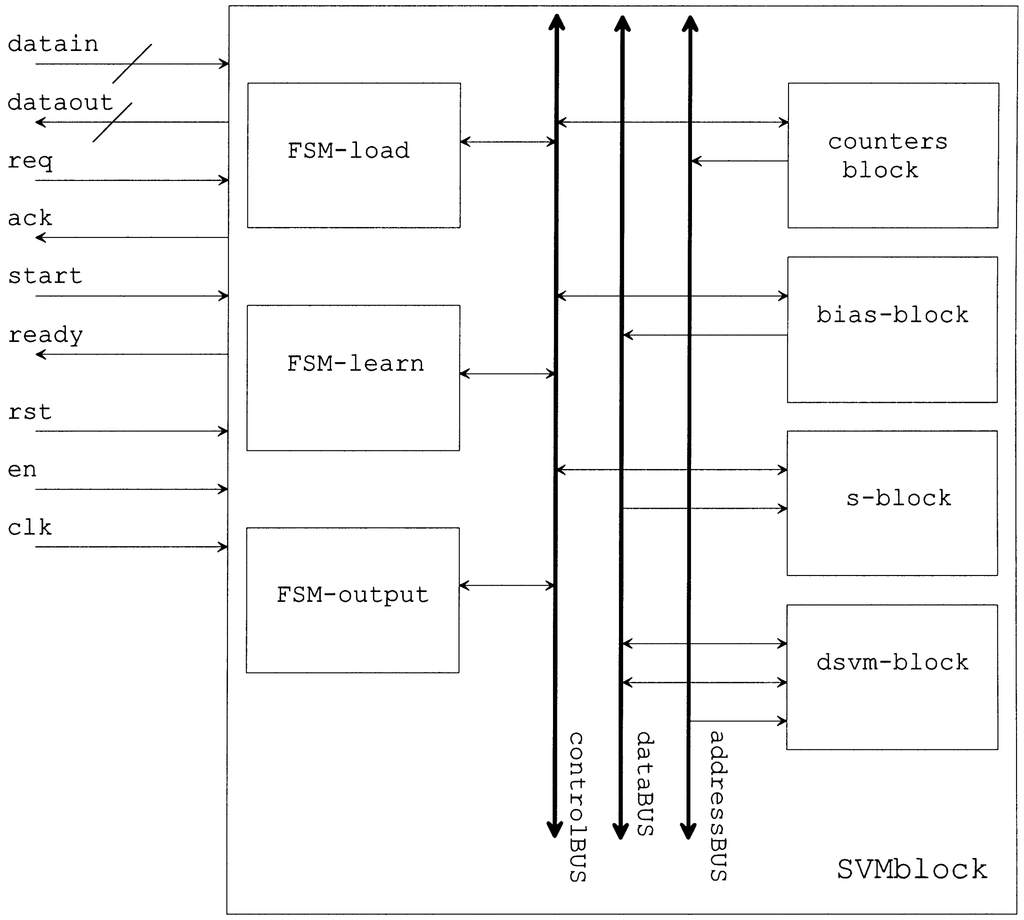


Fig. 1. SVMblock.

force the separating hyperplane to pass through the origin in the feature space and the equality constraint disappears from the dual formulation of the CQP. Then, the DSVM can be used for solving the resulting problem

$$\min_{\boldsymbol{\alpha}} \left[\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \boldsymbol{r}^T \boldsymbol{\alpha}\right]$$
$$0 \leq \boldsymbol{\alpha} \leq C. \tag{16}$$

The core of DSVM is very simple

$$\boldsymbol{z}^k = \boldsymbol{\alpha}^k + \eta \boldsymbol{g} \tag{17}$$
$$\boldsymbol{\alpha}^{k+1} = \max\left(0, \min\left(\boldsymbol{z}^k, C\right)\right) \tag{18}$$

where $\boldsymbol{g} = -Q\boldsymbol{\alpha} + \boldsymbol{r}$. The convergence of the above recurrent relation is guaranteed, provided that $\eta \leq 2/m$.

Even if several experiments on real-world data sets have demonstrated the effectiveness of this method, it is greatly penalized by two facts: the generalization of a SVM with a constant threshold does not fit the usual theoretical framework, and it is not easily applicable to other kernels (e.g., the linear one).

Here, we suggest a new approach, which allows the use of the DSVM algorithm and the computation of the threshold as well. This result is based on a recent work on parametric optimization [18]: the main idea is to design a learning algorithm composed by two parts, working iteratively. The first part addresses the resolution of a CQP with fixed b, whereas the second one implements a procedure for updating the value of b itself, in order to reach iteratively the optimal threshold value b^*.

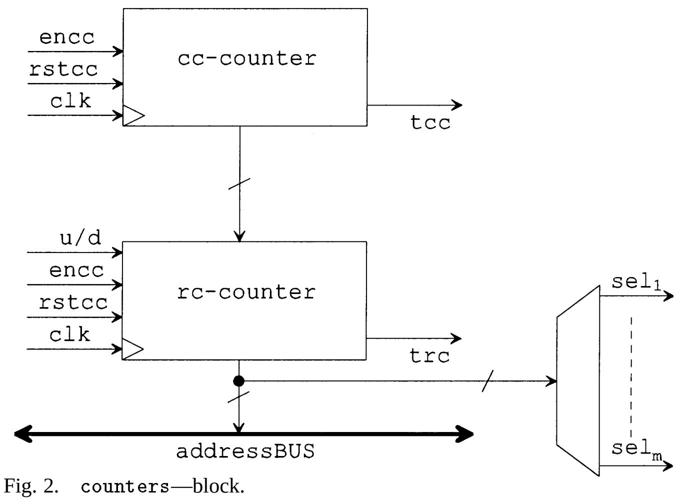


Fig. 2. counters—block.

If we consider the $\mathcal{P}$ problem of SVM learning, it is easy to deduce that, whenever the threshold b is considered as an *a priori* known parameter, then the dual formulation becomes

$$\min_{\boldsymbol{\alpha}} \left[\frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \tilde{\boldsymbol{r}}^T \boldsymbol{\alpha}\right]$$
$$0 \leq \boldsymbol{\alpha} \leq C. \tag{19}$$

where $\tilde{\boldsymbol{r}} = (\boldsymbol{r} - \boldsymbol{y}b)$.

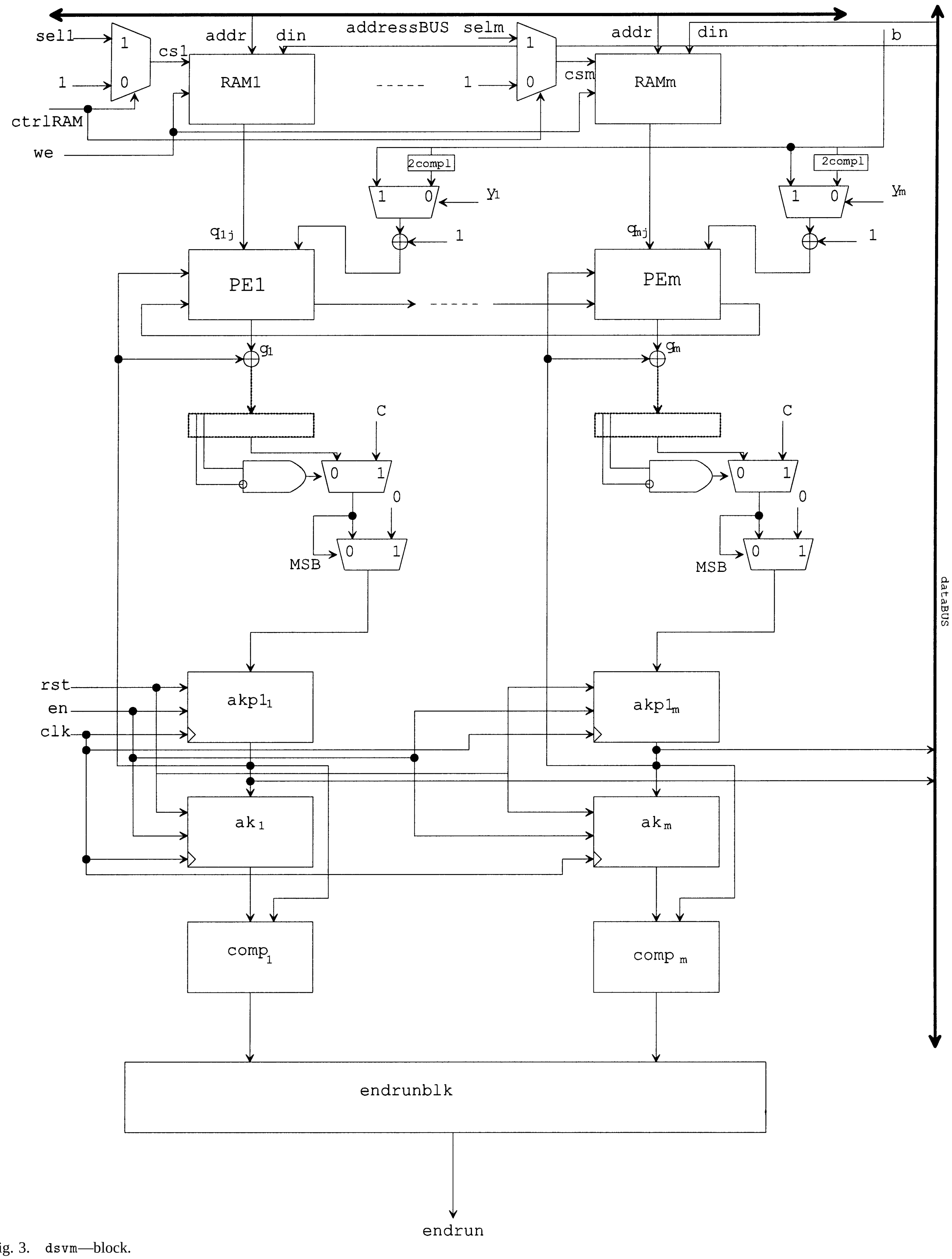


Fig. 3. `dsvm`—block.

Such a problem can be solved by a slightly modified version of DSVM, listed in Table I. Given the input kernel matrix Q, a threshold b, and a starting value $\boldsymbol{\alpha}^0$, the algorithm solves the CQP of (19) by providing an intermediate solution $\boldsymbol{\alpha}'$.

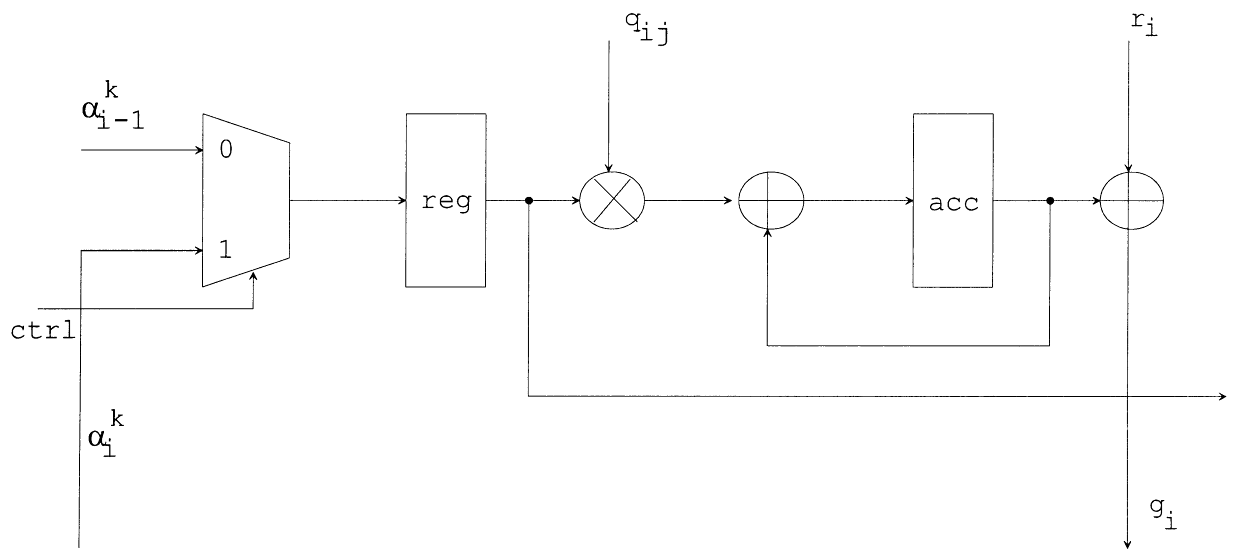


Fig. 4. Generic PE.

As our aim is to solve the CQP with the equality constraint, it is possible to deduce, from the term $s = \text{sgn}\left(\boldsymbol{y}^T\boldsymbol{\alpha}'\right)$, the range of variation of b^*. In fact, being b^* the optimal value of the threshold to be find and b a tentative value at a given step, if $s < 0$ then $b^* < b$, otherwise if $s > 0$ then $b^* > b$: in other words, s is a nonlinear function of b that crosses zero only once, when $b = b^*$, therefore, b lies in a range $[b_{low}, b_{up}]$ for which $s_{low} > 0$ and $s_{up} < 0$ [18]. Consequently, a simple bisection procedure can be derived for finding b^*, that is the only point where $s = 0$. Our proposal, called *FIxed b Svm* (Fibs), is listed in Table II: it has been designed with the goal of an implementation on a digital architecture, but, as will be clear subsequently, it can be implemented on a general-purpose floating-point platform as well.

Its functionality is based on the search of a range of values $[b_{low}, b_{up}]$ to which the threshold b^* belongs. Then, at each step, it proceeds according to a simple bisection process by finding a tentative value $b = (b_{low} + b_{up})/2$, and by updating b_{low} and b_{up} on the basis of the value of s. It terminates when the range $[b_{\text{low}}, b_{\text{up}}]$ becomes smaller than a given tolerance ε_b. Note that, when the algorithm starts, both b_{low} and b_{up} are not known, therefore, a first search of the feasible range must be performed. We found experimentally that $b_{\text{low}} = -1$ and $b_{\text{up}} = 1$ are good starting choices.

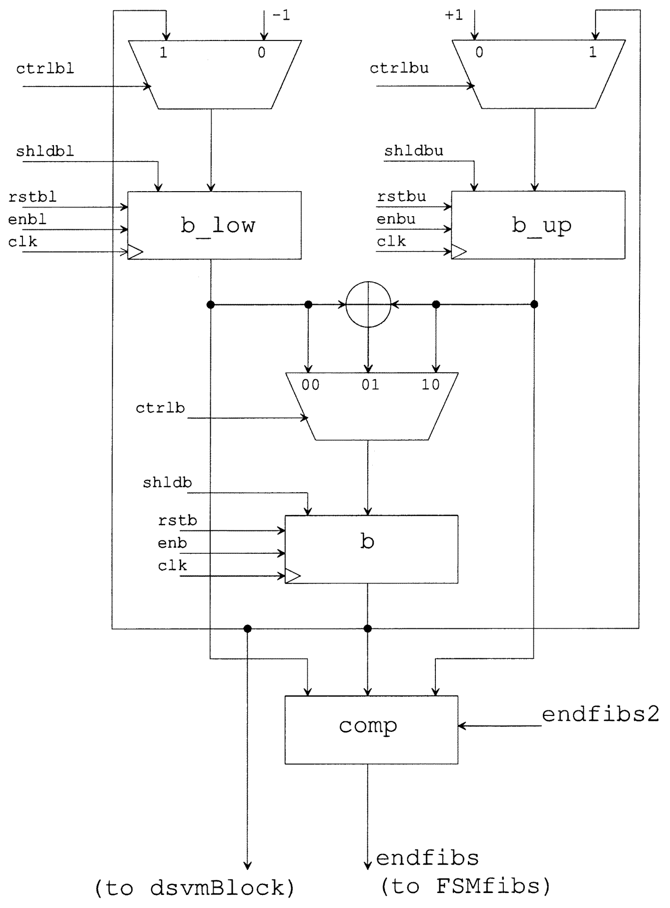


Fig. 5. bias–Block.

B. Quantization Effects in SVMs

To the best of our knowledge, no theoretical analysis of the quantization effects of the SVM parameters has appeared so far in the literature. For this reason, we quantify here some of these effects in order to prove that the feedforward phase of the SVM can be safely implemented in digital hardware.

We perform a worst-case analysis based on the properties of interval arithmetic [13], [11]. Other techniques can be used as well (e.g., statistical methods [12]), but we believe that worst-case results, when applicable, can be of more interest, when targeting digital architectures, because provide guaranteed bounds on the performance of the implementation.

The effect of quantization can be described by transforming each variable involved in the computation of the feedforward phase in an interval

$$\alpha_i \rightarrow [\alpha_i - \Delta\alpha, \alpha_i + \Delta\alpha] \tag{20}$$

$$k_i \rightarrow [k_i - \Delta k, k_i + \Delta k] \tag{21}$$

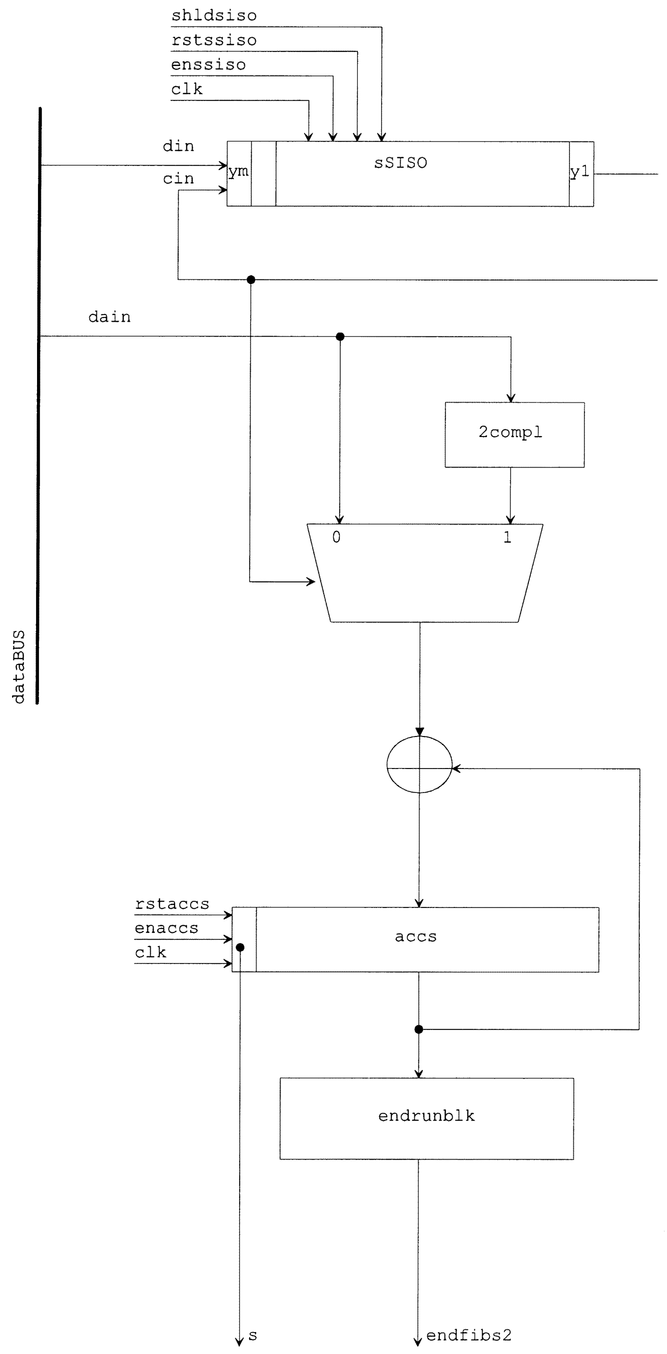


Fig. 6. s-block.

$$b \to [b - \Delta b, b + \Delta b] \tag{22}$$

where $\Delta\alpha$, Δk, and $\Delta b \geq 0$ are no less than half of the quantization step. Note that, for simplifying the notation $k_i = k(\boldsymbol{x}, \boldsymbol{x}_i)$.

Using the properties of interval arithmetic [13] and the fact that $\alpha_i \geq 0$, it is possible to derive upper bounds of the quantization effects

$$\left[\hat{\Phi}(\boldsymbol{x}) - \Delta\Phi, \hat{\Phi}(\boldsymbol{x}) + \Delta\Phi\right] \tag{23}$$

$$= \sum_{i=1}^{m} y_i [\alpha_i - \Delta\alpha, \alpha_i + \Delta\alpha] \times [k_i - \Delta k, k_i + \Delta k] + [b - \Delta b, b + \Delta b] \tag{24}$$

TABLE III
MAIN CHARACTERISTICS OF FSMs

FSM	States	Transitions	inputs	outputs
FSMload	14	31	4	17
FSMout	5	11	3	9
FSMFibs	14	34	5	12
FSMs	5	7	2	8
FSMdsvm	6	9	3	14

$$\subseteq \sum_{i=1}^{m} y_i [\alpha_i k_i - \alpha_i \Delta k - k_i \Delta\alpha + \Delta\alpha\Delta k, \alpha_i k_i + \alpha_i \Delta k + k_i \Delta\alpha + \Delta\alpha\Delta k] + [b - \Delta b, b + \Delta b] \tag{25}$$

$$\subseteq \sum_{i=1}^{m} y_i \alpha_i k_i + b + \Delta\alpha\Delta k \times \sum_{i=1}^{m} y_i + \sum_{i=1}^{m} y_i [-\alpha_i \Delta k - k_i \Delta\alpha, +\alpha_i \Delta k + k_i \Delta\alpha] + [-\Delta b, \Delta b] \tag{26}$$

$$\subseteq \hat{\Phi}(\boldsymbol{x}) + \left(m^+ - m^-\right)\Delta\alpha\Delta k + m[-C\Delta k - \Delta\alpha, C\Delta k + \Delta\alpha] + [-\Delta b, \Delta b] \tag{27}$$

where m^+ (m^-) is the number of patterns with positive (negative) target and $m^+ + m^- = m$. In the above derivation, we have assumed, for simplicity, that $k_i \leq 1$, as in Gaussian kernels, but a similar result can be derived for other kernels as well.

Then, the bound on the quantization effect is given by

$$[-\Delta\Phi, +\Delta\Phi] \subseteq \left(m^+ - m^-\right)\Delta\alpha\Delta k + (m(C\Delta k + \Delta\alpha) + \Delta b)[-1, +1]. \tag{28}$$

The above equation can be used to analyze the contribute to the quantization error from each parameter of the SVM. As an example of this analysis, let us assume that each parameter is quantized with a similar step $\Delta\alpha = \Delta k = \Delta b = \Delta$. According to SVM theory, any pattern lying outside the margin gives an output $|\hat{\Phi}(\boldsymbol{x})| \geq 1$; therefore, a pattern can be misclassified if the total quantization error is greater than one and of opposite sign respect to the correct target value. In other words, using (28), a necessary condition for a misclassification error is given by

$$\left(m^+ - m^-\right)\Delta^2 - m\Delta(C+1) - \Delta \geq -1. \tag{29}$$

If the two classes are balanced $(m^+ = m^-)$ the above equation simplifies to an intuitive relation between the maximum admissible quantization step and some parameters of the SVM

$$\Delta \leq \frac{1}{m(C+1)+1}. \tag{30}$$

Equation (30) shows that the quantization step cannot be greater than a quantity that depends on: 1) the number of the patterns and 2) the size of the alphas. This is an intuitive result: in fact, as the number of patterns grows, the summation on them can suffer from the accumulation of quantization errors

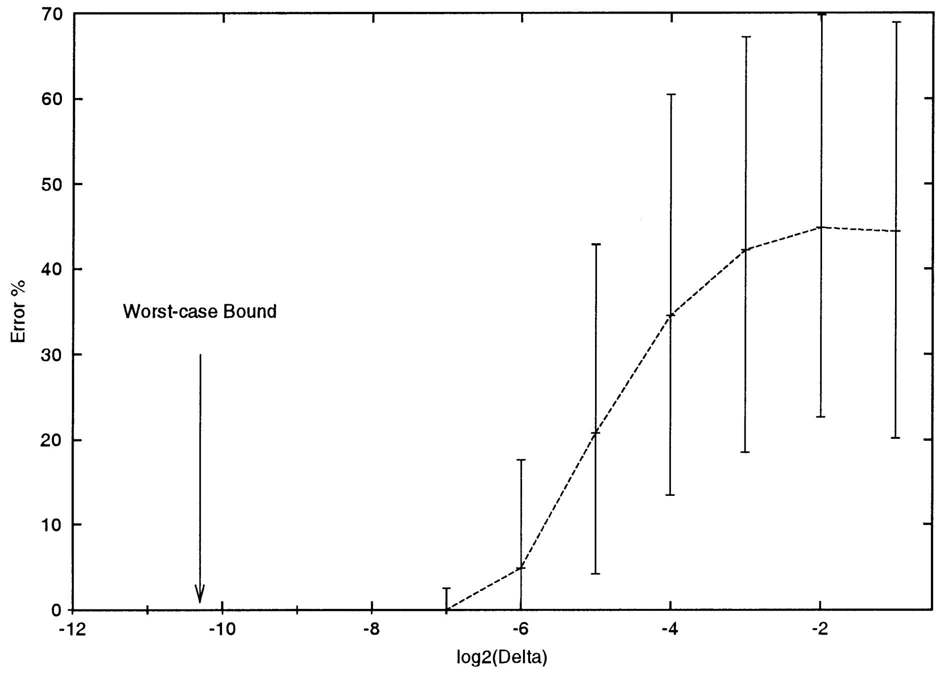


Fig. 7. Misclassification error due to quantization effects.

and, at the same time, larger values of the parameters can heavely affect the classification output.

IV. Digital Architecture

In Section III, we have detailed the Fibs algorithm and we have shown that can it be applied to solve an SVM-based classification problem with any kind of kernel function. In this section, we describe in detail an architecture that implements it, following a top-down approach. At first, we will show the main signals and blocks that compose the design, then we will detail each one of them. The architecture described in this paper is just a first study on this topic, useful to understand its main properties, such as number of clock cycles needed to reach a feasible solution, clock frequency, device utilization, etc. With this aim, we focus our attention on the design of a prototype of a RBF-SVM, but the same design can be easily extended to different kind of SVMs.

A. Main Blocks and Signals

The input–output interface of our design is represented in Fig. 1. It is characterized by two data lines, namely `datain`, used for input data, and `dataout`, used for the output, of 16 bits each. The lines `req` and `ack` act as handshake signals for the I–O. The signal `start` begins a learning session, while the `ready` signal reports both its termination and the fact that the entity can start to submit the parameters b and $\boldsymbol{\alpha}$ to the output. Finally, `en`, `rst`, `clk` are the enable, the asynchronous reset, and the clock signals, respectively.

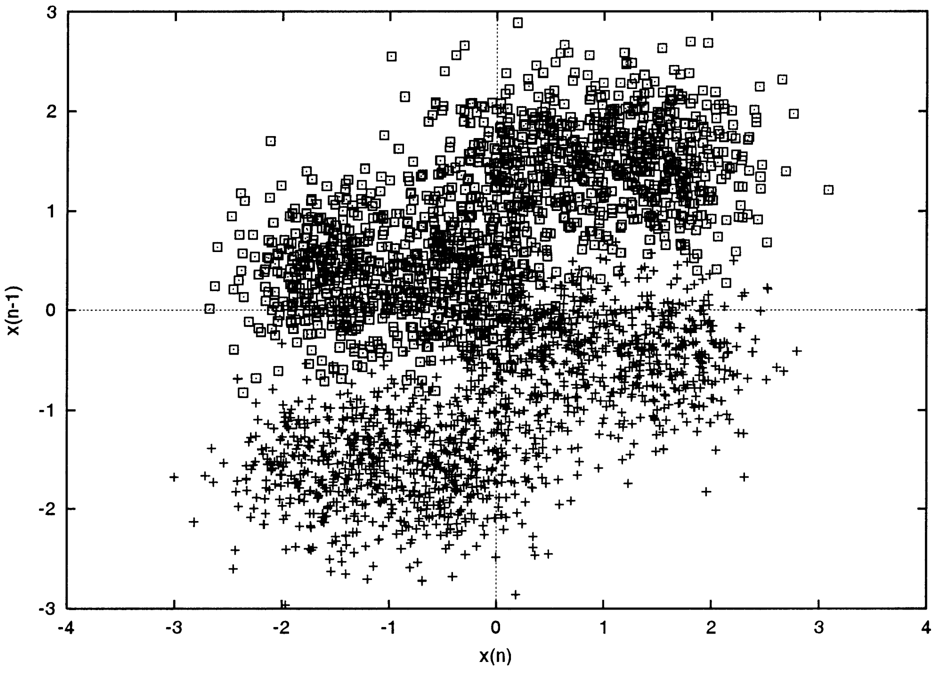
The functionality of the `SVMblock` can be subdivided in three basic phases.

1) *Loading Phase*: When the SVMblock receives a `start`, transition $0 \rightarrow 1$, it begins the loading of the target vector $\boldsymbol{y}$, and the kernel matrix $\boldsymbol{Q}$. Actually, without loss of generality, we suppose that the SVMblock receives and stores in its memory directly the negated $\tilde{\boldsymbol{Q}} = -\boldsymbol{Q}$. The values must be delivered by scanning the matrix row by row (or column by column, thank to the simmetry properties of $\boldsymbol{Q}$), and following a typical handshake protocol. The activation of the asyncronous reset permits one to clean up all registers and to start a new learning with the initial value $\boldsymbol{\alpha} = 0$, otherwise the final value of the previous learning is assumed as new starting point.
2) *Learning Phase*: As soon as the loading is completed, the `SVMblock` starts the learning phase, according to the *Fibs* algorithm detailed in Section III; once it terminates, the ready signal is activated in the output, and the block can begin the output phase;
3) *Output phase*: The `SVMblock` submits the results of learning, that is the values b^*, $\alpha_1^*, \ldots, \alpha_m^*$, to the output using the same communication protocol of the loading phase.

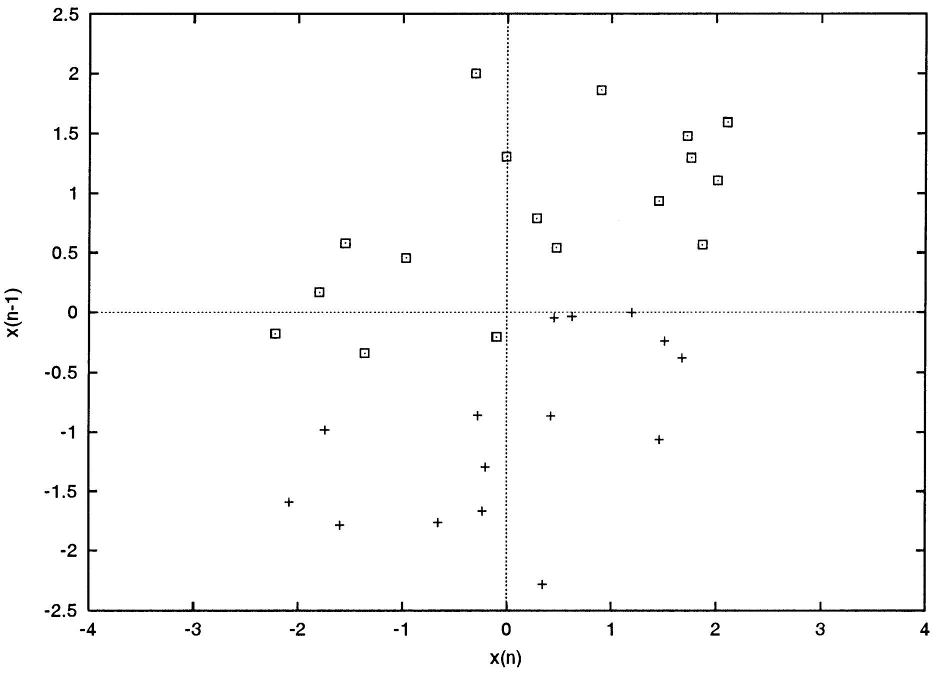
These logical phases are implemented by the general architecture depicted in the block-scheme of Fig. 1. It is mainly composed by four computing blocks, namely the `counters`, `dvsm`, `bias`, and `s`-blocks, and three controllers for the loading, learning, and output phase, respectively. Whereas all the signals to/from the controllers are connected on the `controlBUS` via a tristate-based connection, data are connected on the `dataBUS`, while the information on the `addressBUS` indexes each element of the kernel matrix, as detailed in the following.

B. Structure of Each Block

In this section, we will describe the function of each basic block of Fig. 1, each of which implements a specific function expressed in algorithm *Fibs*: the `dvsm`—block contains both the memory to store matrix $\tilde{\boldsymbol{Q}}$ and all the digital components needed to implement the algorithm of Table I; the `counters`-block provides all the counters and indexes needed to select either the entries of the kernel matrix and each alpha value during the flow of



(a)



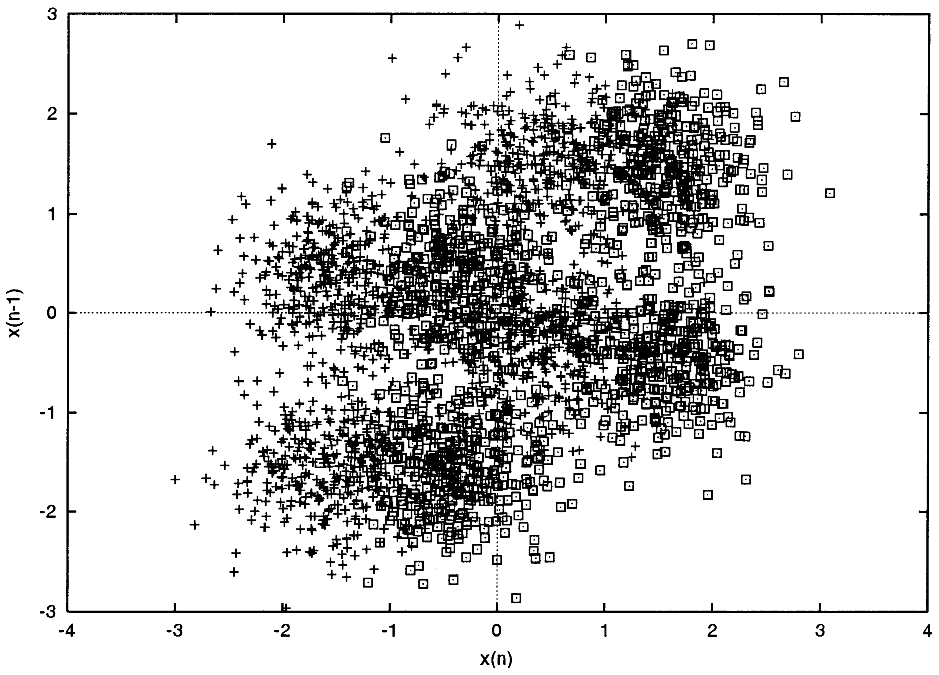
(b)

Fig. 8. Distribution of data for Models 1a (a) and 1b (b) $[D = 2, \sigma_e^2 = 0.2, m = 500(A), m = 32(B)]$.

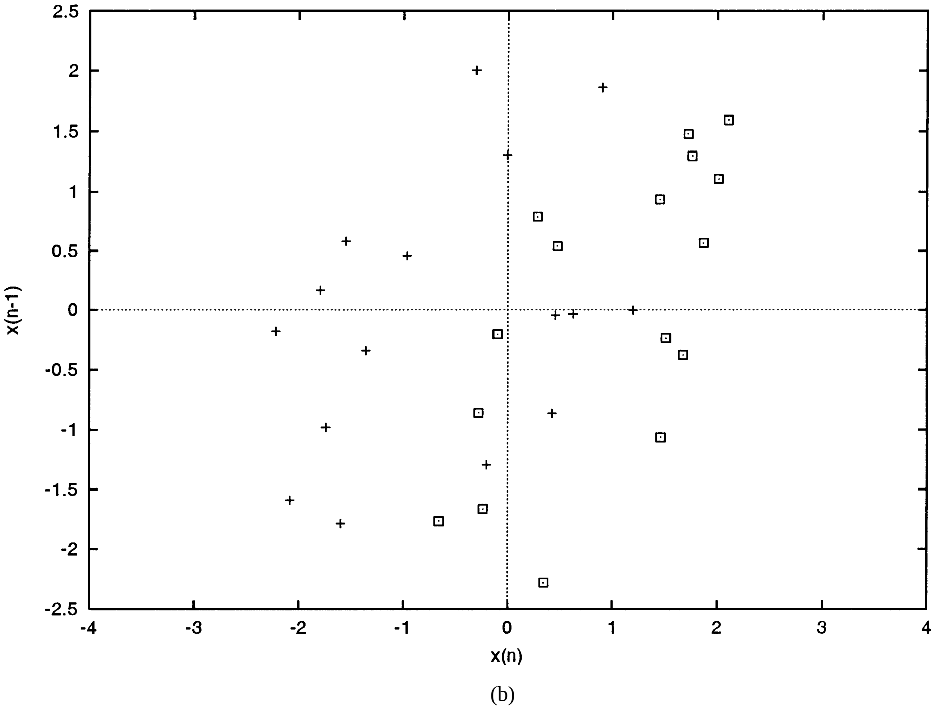
the algorithm; the `bias`-block contains registers and combinatorial logic for the control the values $b_{\rm low}$, $b_{\rm up}$, and b. Finally, the `s`-block contains a register where the target vector $\boldsymbol{y}$ is stored, and the logic components for the computation of the equality constraint, namely the value of s in Table II. As a final remark, let us note that all the connections to the `controlBUS` and the `dataBUS`, are realized via a tristate connection.

1) `Counters`-*Block:* The structure of the `counters`–block is represented in Fig. 2. It mainly consists of two counters, that we call `cc` (column counter, used to index a column of the kernel matrix) and `rc` (row counter, used to index a row of the kernel matrix), respectively. The `cc` is also used to select a particular RAM, during loading, inside the `RAM`-block of `dvsm`, and to select a particular value of alpha, both during the computation of s and during the output phase.

The output of `rc` is directly connected to the `addressBUS`. The cc is an up-counter, while the rc s counts both up and down, as detailed in Section IV-B2.



(a)



(b)

Fig. 9. Distribution of data for Models 2a (a) and 2b (b) $[D = 0, \sigma_e^2 = 0.2, m = 500(A), m = 32(B)]$.

2) `Dvsm`*-Block:* We implemented the DSVM algorithm in a one-dimensional systolic array composed by m basic processing elements (`PE` s), each of which has the task of updating an element of the vector $\boldsymbol{g}$. As previously indicated, we suppose that matrix $\tilde{\boldsymbol{Q}}$ is precomputed and stored in a set of RAMs of size $m \times N_Q$ bit; each RAM contains a row of $\tilde{\boldsymbol{Q}}$, as shown in Fig. 3. N_Q is the length of the word used to code each element $\tilde{q}_{ij}$, while with N_α we indicate the length of the words used to code each α_i. Fig. 4 shows the structure of a generic PE_i.

At a given time-step k, each PE_i holds the component α_i^k in the register `reg`, and its role is to compute $g_i = \sum_j \tilde{q}_{ij}\alpha_j^k + \tilde{r}_i$; the value $\tilde{r}_i$ is known *a priori* at each PE_i, when the `dvsm`-block is enabled to operate, and is deduced from the value b, obtained from the `bias`-block and from the value of the target y_i, provided by `s`.

As a first step, the value $\tilde{q}_{ii}\alpha_i^k + 0$ is computed and stored in `acc`; α_i^k is then delivered to PE_{i+1}, whereas PE_i receives α_{i-1}^k from PE_{i-1}. As a second step, PE_i computes and stores

TABLE IV
FLOATING AND FIXED-POINT EXPERIMENTS FOR SONAR (LINEAR KERNEL)

	nsv	b	eq.	TR	TS
FP(SMO)	45	-16.8489	-2.2E-13	0	23
FP(FIBS)	45	-16.8412	7.8E-4	0	23
16–11–9	60	-4.64	0.13	0	19
16–11–13	46	-15.84	-1.83E-3	0	23
12–11–9	60	-4.7	-1.7E-1	0	20
12–11–13	43	-15.6	-2.8E-2	0	24
8–11–9	48	-153.439	2.9E-2	19	30
8–11–13	48	-149.7	1.8E-3	18	30

TABLE V
FLOATING AND FIXED-POINT EXPERIMENTS FOR MODEL 1A
$(\sigma^2 = 0.5; C = 0.05)$

	nsv	b	eq.	TR	TS
FP(SMO)	271	1.3E-16	1.32E-16	17	88
FP(FIBS)	271	4.29E-2	-2.17E-4	17	88
16–3–8	500	1.17E-2	-3.9E-3	19	90
16–3–13	282	4.45E-2	-6.1E-5	18	89
12–3–8	500	1.17E-2	-3.9E-3	19	90
12–3–13	282	4.45E-2	2.4E-4	18	89
8–3–8	500	1.17E-2	0.0	19	91
8–3–13	282	4.41E-2	1.2E-4	18	88
4–3–8	500	1.17E-2	-1.56E-2	18	88
4–3–13	279	1.17E-2	6.1E-5	18	88

TABLE VI
FLOATING AND FIXED-POINT EXPERIMENTS FOR MODEL 1B
$(\sigma^2 = 0.5; C = 0.09)$

	nsv	b	eq.	TR	TS
FP(SMO)	23	-7.37E-2	-6.9E-18	1	122
FP(FIBS)	23	-7.37E-2	-9.16E-5	1	122
16–3–8	23	-7.4E-2	-1.95E-2	1	124
16–3–13	23	-7.36E-2	0.0	1	122
12–3–8	23	-7.4E-2	-1.56E-2	1	124
12–3–13	23	-7.35E-2	-4.8E-4	1	122
8–3–8	23	-8.2E-2	1.95E-2	1	125
8–3–13	23	-7.54E-2	3.6E-4	1	121
6–3–8	23	-6.6E-2	1.95E-2	1	120
6–3–13	23	-6.12E-2	-3.6E-4	1	118

TABLE VII
FLOATING AND FIXED-POINT EXPERIMENTS FOR MODEL 2A
$(\sigma^2 = 0.5; C = 0.02)$

	nsv	b	eq.	TR	TS
FP(SMO)	324	6.01E-2	0.0	80	386
FP(FIBS)	324	6.01E-2	-3.6E-4	80	386
16–3–8	500	4.3E-2	-7.8E-3	80	386
16–3–13	327	5.69E-2	1.83E-4	80	385
12–3–8	500	4.3E-2	-7.8E-3	80	387
12–3–13	327	5.6E-2	3.6E-4	80	385
8–3–8	500	4.3E-2	-7.8E-3	81	387
8–3–13	329	5.56E-2	-4.2E-4	79	383
6–3–8	500	4.3E-2	3.52E-2	79	392
6–3–13	328	5.76E-2	-3.05E-2	83	388

$\tilde{q}_{i(i-1)}\alpha_{i-1}^k + \tilde{q}_{ii}\alpha_i^k$ and so on. Finally, after m steps, corresponding to $2 \cdot m$ clock cycles, the final value g_i^k is ready at the ouput of each PE_i.

The storing of $\tilde{\boldsymbol{Q}}$ inside the `RAM`-block is the following: each $\mathtt{RAM}_i$ contains the ith row of $\tilde{\boldsymbol{Q}}$, that is the value $\tilde{q}_{ii}$ (stored at location 0), $\tilde{q}_{i(i-1)}$ (stored at location 1), and so on; finally $\tilde{q}_{i(i+1)}$ is stored at location m. In this way the corresponding PE_i has the value $\tilde{q}_{ij}$ ready for the computation, each time a new α_j arrives.

During learning, $\mathtt{RAM}_i$ is indexed by the address given by the counter `rc`, which counts in mode `up`; instead, during loading, the same counter, used to store the ith row, counts in mode `down`, and begins its counting from the value i. Such an initial value is provided by the external counter `cc` (see Fig. 2). Once each component of $\boldsymbol{g}$ is ready, as we suppose here to implement an RBF-SVM, and being for such machines $\max q_{ij} = 1$, a shift of $p = \lceil \log_2 m \rceil - 1$ positions must be executed. To perform the shift, we directly connect the $N_\alpha - p$ most significant wires of each g_i to the less significant wires of the corresponding adder, thus avoiding, in practice, an actual shift. The set of multiplexers subsequent to the adder act as limiters, in order to constraint each alpha to lie inside the box $[0, C]$ and to compute α_i^{k+1}, which is stored in the corresponding register `akp1`. As soon as the value α_i^{k+1} is computed and stored in `akp1`, its previous value α_i^k is stored in `ak`: this further set of registers is useful in order to verify the termination of the algorithm, represented by the activation of the signal `endrun`. In particular, each component `comp` is a simple binary comparator providing "1" when the inputs are all equal and "0" otherwise. Finally, the `endrunblk`-block, implements an AND operation of its inputs.

If a further iteration must be executed $(\texttt{endrun} = 0)$, another cycle begins after the storing of each akpl in the corresponding reg of the PE.

3) Bias-Block: The bias—block (see Fig. 5), is simply composed by three left-shift registers, containing b_{low}, b_{up}, and b, an adder, two 2-to-1 multiplexers, a 3-to-1 multiplexer, and a binary comparator, which detects the termination of the learning phase $(\texttt{endfibs} = 1)$ whenever $b = b_{\text{low}}$ or $b = b_{\text{up}}$. Note that, as $\boldsymbol{y}^T\boldsymbol{\alpha}' = 0$ is a feasible condition for stopping the algorithm, the bias-block receives also the signal endfibs from the s-block, which is equal to "1" when $\boldsymbol{y}^T\boldsymbol{\alpha}' = 0$ and "0" otherwise. Finally, the bisection process $(b = (b_{low} + b_{up})/2)$ is simply implemented without connecting the LSB of the adder to the input of the multiplexer (such a connection is represented with a dashed line in Fig. 5), thus avoiding, as in the case of the multiplication for η discussed in Sections IV-B1 and IV-B2, an actual right-shift.

4) S-Block: The role of the S–block (see Fig. 6), is the computation of s, namely the sign of the quantity $\boldsymbol{y}^T\boldsymbol{\alpha}'$. It is composed by a serial-input serial-output circular shift register (sSISO), containing the vector $\boldsymbol{y}$, a "2's complement" block, a 2-to-1 multiplexer, an adder and a register (accs) acting as an accumulator. The value s is simply the MSB of accs. The s-block works during the loading phase, when the labels $y_1, \ldots, y_m$ are stored in the sSISO, and during the learning phase after the value $\texttt{endrun} = 1$ is generated by the dvsm-block. The connection with the databus is as follows: the serial-input din of sSISO, read only when the input shldSISO is active high (load), is connected to the LSB of dataBUS, while the vector dain is fully connected to dataBUS. In practice, dain contains, at a given time during learning, a value α_i, from the set of registers akpl.

5) Controllers: As previously indicated, the SVMblock is composed by three finite-state machines (FSMs), each one having the role of controlling a given phase of the Fibs-algorithm; we indicate them as FSMload, FSMlearn and FSMoutput, respectively. Actually, FSMlearn is subdivided in three different modules that we call FSMfibs, FSMdsvm and FSMs. FSMfibs is the actual supervisor of the architecture and manages all the transitions for the correct flow of algorithm Fibs. In particular, it enables FSMdsvm to control the functionality of the dsvm-block, during a part of learning, and, after the value $\texttt{endrun} = 1$ is obtained, enables the submodule FSMs, which supervises the s-block. FSMfibs receives all the control signals endrun, s and endfibs, and decides the correct action according to their value, following the algorithm given in Table II. Table III, reports the main characteristics of the FSMs, such as number of states, transitions, total inputs/outputs. The choice of subdividing the controller of the system in five different sub-modules, is justified by the fact that, for synthesis purposes, it is better to have small modules working separately, as confirmed by the results discussed in Section V-D.

V. Experimental Results

In order to test the proposed algorithm and the corresponding digital architecture, we chose a well-known benchmarking dataset (Sonar), and several datasets from a telecommunication problem, recently used for the application of SVMs to channel equalization purposes. In Section V-A, we also show how the theoretical results on the quantization effects (Section III-B) relate to the actual misclassification of the SVM on the Sonar dataset. The second problem, described in Section V-B, is particularly appealing because it is a typical case where a dedicated hardware can be of great usefulness. The floating and the fixed-point experiments are discussed in Section V-C; furthermore, in such section we report the comparison of our approach with the well-known SMO algorithm [9], [19], [20] for SVM learning. Finally, in Section V-D, we discuss the implementation on a FPGA.

TABLE VIII
Floating and Fixed-Point Experiments for Model 2b
$(\sigma^2 = 0.5; C = 0.08)$

	nsv	b	eq.	TR	TS
FP(SMO)	28	2.05E-2	0.0	3	683
FP(FIBS)	28	2.06E-2	-1.2E-4	3	683
16–3–8	29	2.73E-2	-1.17E-2	3	691
16–3–13	28	2.07E-2	-1.2E-4	3	685
12–3–8	29	2.7E-2	.1.17E-2	3	692
12–3–13	28	2.09E-2	-3.6E-4	3	685
8–3–8	29	2.7E-2	-2.34E-2	3	692
8–3–13	28	1.97E-2	-3.05E-4	3	688
4–3–8	28	2.73E-2	-7.8E-3	5	685
4–3–13	28	3.46E-2	-3.6E-4	3	680

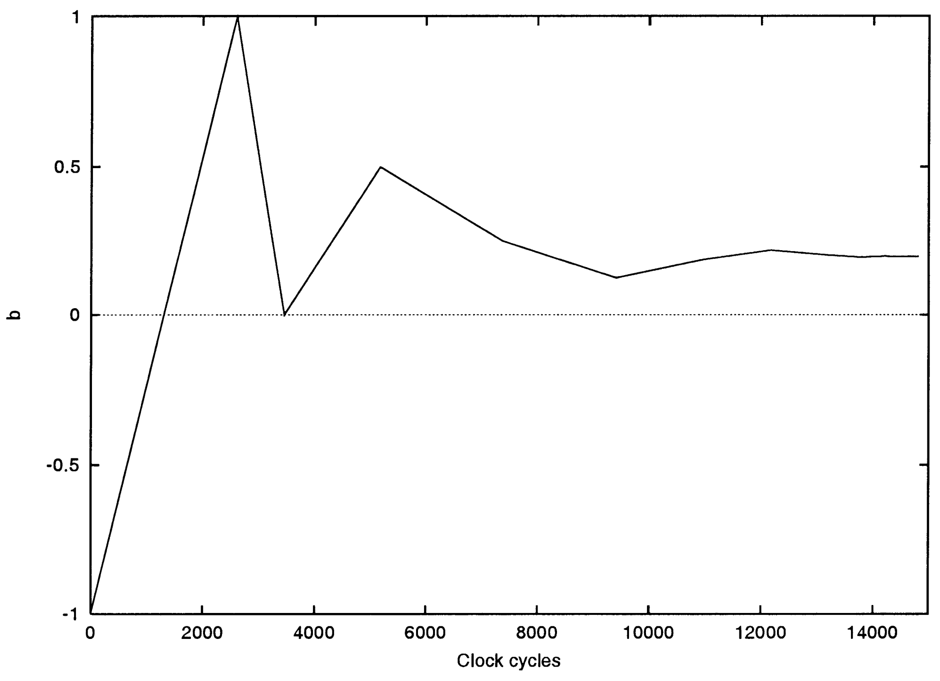
A. Sonar Dataset

The Sonar dataset is a well-known linearly separable problem, extensively used for benchmarking purposes of learning algorithms [21]: its popularity is mainly due to the difficulty of the classification task. As the computation of the threshold b is fundamental for the determination of the separating surface, it is particularly suitable to test the algorithm proposed in this paper. The sonar data set is composed by 208 samples of 60 features each, usually subdivided in 104 training and 104 test patterns. It is known that a linear classifier misclassifies 23 test patterns, while a RBF-SVM misclassifies six test patterns if the threshold is used, and eight test patterns otherwise [17].

The Sonar dataset has been used for testing the quality of the quantization error bounds found in Section III-B. In particular, we solved the problem using a Gaussian kernel with $\sigma^2 = 1$ and $C = 10$: the solution consists of $m = 119$ support vectors $(m^+ = 59,\, m^- = 60)$. Then, a random perturbation of size Δ has been applied to each parameter of the network and the average, minimum, and maximum misclassification errors have been registered on 10^6 different trials.

TABLE IX
HDL SYNTHESIS REPORT FOR $m = 8$ AND $m = 32$

Block	m	Registers	Counters	Mux	Tristates	Decoders	Add/Sub	Comps
s	8	9	–	2	–	–	2	–
counters	8	4	1	–	–	1	2	–
bias	8	48	–	–	48	1	1	2
dsvm	8	32	–	40	1	–	40	8
s	32	33	–	2	–	–	2	–
counters	32	4	1	–	–	1	2	–
bias	32	48	–	–	48	1	1	2
dsvm	32	128	–	160	1	–	160	32
SVM	8	93	1	90	20	1	45	10
SVM	32	213	1	210	20	1	165	34



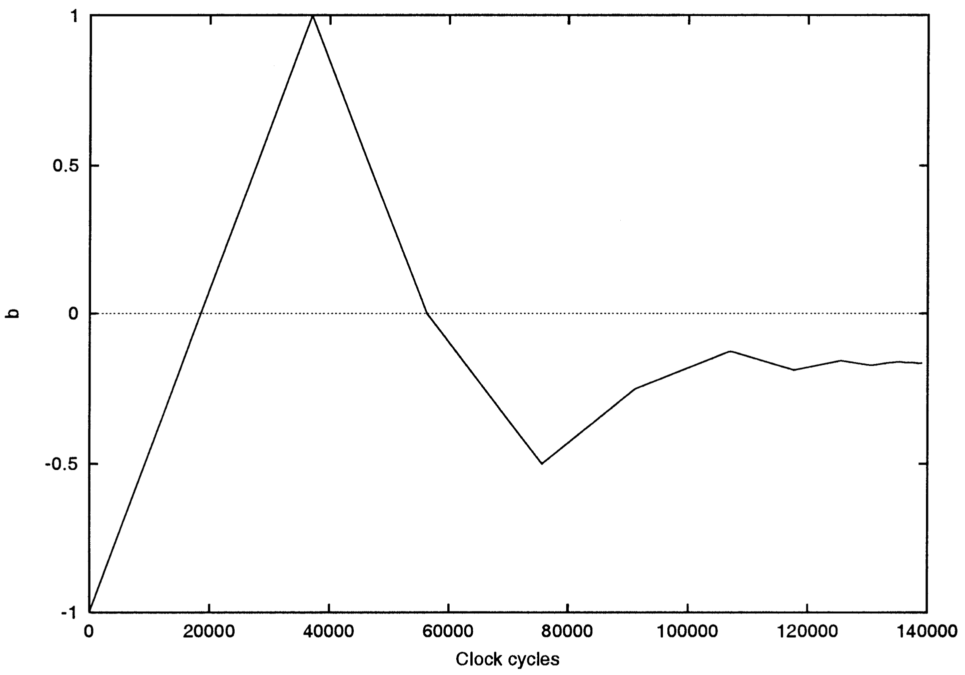
(a)

Fig. 10. Convergence of b for (a) $m = 8$.

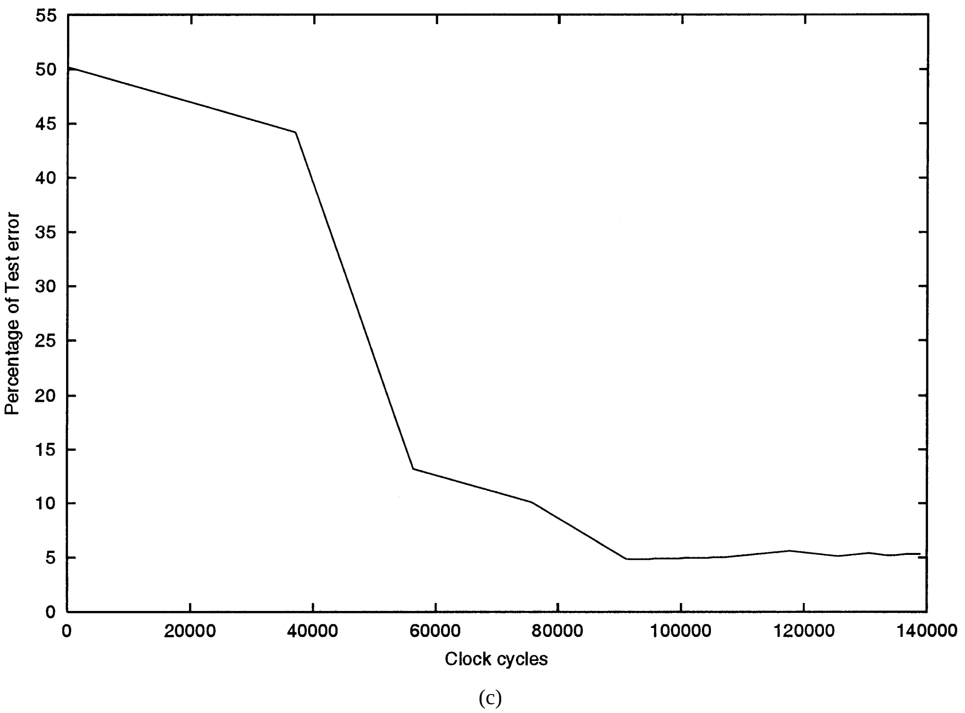
The results are shown in Fig. 7, as a function of $\log_2(\Delta)$: the worst-case bound given by (29), as expected, is quite conservative and suggests that approximately ten bits are necessary for avoiding any misclassification error, while the experimental value is eight. However, it is worthwhile noting that 10^6 trials are an infinitesimal amount respect to an exhaustive search for misclassification errors, but require many CPU hours on a conventional PC. The actual number of possibilities of adding $\pm\Delta$ to each parameter of the SVM is 2^{2m+1}, therefore, an exhaustive search is impossible to perform in practice and this approach could never provide enough confidence on the result. The worst-case bound, instead, provides a safe value, which can be easily computed.

B. Channel Equalization Problem

The channel equalization problem is a typical application where a special-purpose device can be effectively used, on the receiver side, in order to estimate one between two symbols $u_n \in \{\pm 1\}$, of an independent sequence emitted from a given source. All the unknown nonlinear effects of the involved components (transmitter, channel and receiver) are modeled



(b)



(c)

Fig. 10. *(Continued)* Convergence of b for percentage of test errors during learning for (b) $\mathrm{m} = 32$ and (c) $m = 32$.

as finite-impulse response (FIR) filters, plus a Gaussian distributed noise e with zero mean and variance σ_e^2

$$\begin{aligned}\tilde{x}(n) =& \sum_{k=0}^{N} h_k u(n-k) \\ \hat{x}(n) =& \sum_{p=1}^{P} c_p \tilde{x}^p(n) \\ x(n) =& \hat{x}(n) + e(n). \end{aligned} \tag{31}$$

The classical theory, tackles this problem by finding an optimal classifier (the Bayesian maximum —ikelihood detector), which provides an estimate $\hat{u}(n - D)$ of $y_n = u(n - D)$ through the observation of an l-dimensional vector $\boldsymbol{x}_n = [x(n), x(n-1), \ldots, x(n-l+1)]^T$. Whereas these methods require the knowledge of the symbols probability, and the analytic structure of the model, neural network-based approaches have been successfully applied [22], [23] to systems where such information are not known.

TABLE X
POST-PLACE AND ROUTE SYNTHESIS REPORT FOR DIFFERENT NUMBER OF INPUT SAMPLES AND DIFFERENT OPTIMIZATION CRITERIA

m,criteria	Slices	LUTs	Clock (MHz)
32,area	2950(6%)	4978(5%)	18.9
32,speed	2919(6%)	4916(5%)	21.06
8,area	849(1%)	1457(1%)	34.2
8,speed	865(1%)	1466(1%)	35.3

In practice, a classifier is selected on the basis of m previous samples, having the following structure:

$$\{(\boldsymbol{x}_{n-i}, u(n-D-i))\}_{i=0}^{m-1}. \tag{32}$$

Here, we consider the following two different nonlinear models of the channel, that, substantially, differ for the delay D.

- Model 1: $D = 2$.
- Model 2: $D = 0$.

Furthermore, as our aim is to test the behavior of the algorithm and, above all, to study its actual hardware implementation, we consider different number of training patterns. In particular, we choose $m^a = 500$, as in [23], and $m^b = 32$: this last choice guarantees a good tradeoff between the final generalization ability of the learned model and its device utilization. We call Model 1a, 1b, 2a, 2b, the corresponding distributions. Finally, in order to estimate the generalization error of the obtained SVM, we use a separate test set composed by $m_t^a = 2400$ and $m_t^b = 2900$ samples, respectively. With reference to (31) we consider the following channel:

$$\begin{aligned}\hat{x}(n) &= \tilde{x}(n) - 0.9\tilde{x}^3(n)\\ \tilde{x}(n) &= u(n) + \frac{1}{2}u(n-1)\end{aligned} \tag{33}$$

that assumes an ISI equal to 2. We choose $l = 2$ and $\sigma_e^2 = 0.2$. Figs. 8 and 9 show the distribution of data obtained by (33) with the given parameters.

C. Floating-and Fixed-Point Experiments

When designing a digital architecture one of the most important aspects that must be considered is the length of the word that represents the information inside the design. This parameter has a crucial role because it influences both the length of the registers and, as a consequence, the device utilization and the performance of the digital learning system.

When we faced the design of Fibs and the design of the corresponding architecture in particular, we needed to understand: 1) its behavior when using a floating-point math with respect to standard SVM learning algorithms, like the SMO; 2) the required number of bits. To obtain these answers, we designed several experiments, both on the Sonar dataset, using a linear kernel, and on models 1a, 1b, 2a, and 2b.

The results of our first experiment on the Sonar dataset are reported in Table IV. In this table we report, for every kind of architecture, the number of support vectors (nsv), the value of the threshold (b), the value of the quantity $\boldsymbol{y}^T\boldsymbol{\alpha}'$ (eq.), the number of training errors (TR), and, finally, the number of test errrors (TS). As expected, the floating-point (FP) version of our algorithm obtains good results with respect to the SMO algorithm. Note that we set $\varepsilon = 10^{-6}$ and $\varepsilon_b = 10^{-4}$.

Table IV shows both floating- and fixed-point results for different register lengths. The results of the fixed-point experiments are reported in the second part of the table, where, with the notation $xx - yy - zz$, the following information is indicated: the number of bits used to code each $\tilde{q}_{ij}(xx)$, the number of bits used to code the integer (yy), and the fractional part (zz) of each α_i and b.

From the observation of the results three main important properties emerge: 1) Fibs requires a relatively low number of bits, especially to code the kernel matrix; 2) the accuracy obtained on the equality constraint is not very critical, but, above all; 3) the quantization effect of the kernel matrix is a benefit for the generalization capability. These results are also confirmed by the experiments on the models generated from the telecommunication problem, as reported in Tables V–VIII. In particular, the values reported in the tables clearly indicate that very few bits can be used to code each $\tilde{q}_{ij}$, and that a coding configuration, which outperforms the FP solution, can often be found. Note that we used a RBF-SVM with $\sigma^2 = 0.5$ for both models and $C = 0.05$ (for Model 1a), $C = 0.9$ (for Model 1b), $C = 0.2$ (for Model 2a) and $C = 0.8$ (for Model 2b). To complete our analysis, we compared our results with the ones reported in [23]. The results of our algorithm outperform the ones reported there, obtained with polynomial kernels, even by using only 32 samples. In fact, whereas [23] reaches an accuracy of 4.2% on the test of Model 1a, we reached 3.6% with Model 1a and 4.2% with Model 1b. Similar results are obtained with Model 2: case 2a improves on [23] with an accuracy of 16%; case 2b, instead, is worse, as we measured an accuracy of 23.5%.

D. FPGA Implementation, Functional Simulations, and Synthesis Analysis

The digital architectures described here have been implemented and tested on one of the most powerful Xilinx FPGAs, the Virtex–II, particularly suited to embed high-performance digital signal processors [24]. We chose, as target device, the XC2V8000, characterized by 8M system gates, an array of 112×104 configuration logic blocks (CLBs) providing 46 592 Slices (1 CLB = 4 Slices) and a maximum of 1456 Kbits distibuted RAM. Furthermore, it provides 168 18×18 multiplier blocks and 168 18-Kbit selected-RAM blocks for an amount of 3024 Kbits RAM. Note that the device is very large for current standards and, as detailed in the rest of this paper, only a small amount of hardware is actually used by our architecture.

In order to study the main properties of our design, such as the device utilization, we performed several experiments, at first by choosing a small number of patterns $(m = 8)$, and then by choosing a more realistic size, that is $m = 32$, discussed also, from the generalization point of view, in Section V-C. By using a VHDL description, we could parameterize our design and change the size from $m = 8$ to $m = 32$ without any particular effort, thus allowing an efficient study of the implementation properties for different training set sizes.

Table IX shows the synthesis report, that is, the number of instantiated components for each block and for the whole design.

The functional simulations are summarized in Fig. 10. In particular, it shows the convergence of the threshold b toward the optimum for different number of clock cycles, in both $m = 8$ (A) and $m = 32$ (B) cases. Each learning phase terminates after 14 000 (A) and 140 000 (B) cycles, while each loading phase terminates after 290 (A) and 4226 (B) cycles. Fig. 10(a) and (b) indicates that a feasible b can be reached quite soon during learning: this suggests an acceptable rate of classification can be obtained well before the end of the learning process. In order to validate this assertion, we measured the percentage of test error during learning. Fig. 10(c) shows the value of test errors for different clock cycles for $m = 32$. As one can easily verify, after only 90 000 clock cycles (50 000 before the termination of the learning), the obtained performances are quite stable around the value obtained at the end of the algorithm.

The Xilinx synthesis tool reports an indicative clock frequency, whose final estimated value is known only after the place and route procedure, which physically assigns the resources of the device, such as Slices and connections among them. Table X lists the characteristics of the architecture after the Place and Route phases and shows the device utilization, expressed in number of slices and number of lookup tables, and the clock frequency, for different architectures and optimization criteria. As expected, our approach is particularly efficient from the device utilization point of view. This is a remarkable property because several other modules, such as CPU cores, can be designed on the same device, thus allowing the building of stand-alone intelligent systems on chip.

From the observation of Table X, an interesting and unusual behavior emerges. In fact, in the case $m = 32$, using the speed as optimization criteria, the synthesis tool provides better performances than the area case both in clock frequency and device utilization. An explanation of this could derive from the fact that, when changing the optimization criteria, different synthesis and Place and Route algorithms are applied.

VI. Conclusion

In this paper, we have discussed the implementation of an SVM-based learning approach on a digital architecture and a case-study of its realization on an FPGA device. To the best of our knowledge, this is the first work on this subject, and several issues need to be studied in more detail. First of all, the obtained performances could be greatly improved. In particular, the whole architecture appears to be quite independent from the target device, therefore, it is likely that slightly modified versions exists, which originate mapped-circuits working with higher clock frequencies.

The main point, however, is that an efficient digital architecture can be adopted to solve real-world problems with SVMs, where a specialized hardware for on-line learning is of paramount importance, and it can be easily embedded as a part of a larger system on chip (SoC).

Future work will address this issue, in the spirit of previous neural processors [25], with the integration on the same chip of a general-purpose processing unit and a learning coprocessor module.

Acknowledgment

The authors wish to thank two anonymous reviewers for helping to improve both the structure and the content of the paper and pointing out some imprecision in the first draft of this work.

References

[1] C. S. Lindsey, *Neural Networks in Hardware: Architectures, Products and Applications*. Stockolm, Sweden: Roy. Inst. Technol. On-Line Lectures, 1998.

[2] C. Mead, *Analog VLSI and Neural Systems*. Reading, MA: Addison-Wesley, 1989.

[3] R. Sarpeshkar, "Analog vs. digital: Extrapolations from electronics to neurobiology," *Neural Comput.*, vol. 10, pp. 1601–1638, 1998.

[4] A. R. Omondi, "Neurocomputers: A dead end?," *Int. J. Neural Syst.*, vol. 10, no. 6, pp. 475–482, 2000.

[5] U. Rückert, "ULSI architectures for artificial neural networks," *IEEE Micro Mag.*, pp. 10–19, May–June 2002.

[6] V. Vapnik, *Statistical Learning Theory*. New York: Wiley, 1998.

[7] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*. Cambridge, U.K.: Cambridge Univ. Press, 2000.

[8] B. Schölkopf and A. Smola, *Learning With Kernels*. Cambridge, MA: MIT Press, 2002.

[9] C.-J. Lin, "Asymptotic convergence of an SMO algorithm without any assumption," *IEEE Trans. Neural Networks*, vol. 13, pp. 248–250, Jan. 2002.

[10] P. Laskov, "Feasible direction decomposition algorithms for training support vector machines," *Machine Learning*, vol. 46, pp. 315–350, 2002.

[11] D. Anguita, S. Ridella, and S. Rovetta, "Worst case analysis of weight inaccuracy effects in multilayer perceptrons," *IEEE Trans. Neural Networks*, vol. 10, pp. 415–418, Mar. 1999.

[12] Y. Xie and M. A. Jabri, "Analysis of the effects of quantization in multilayer neural networks using a statistical model," *IEEE Trans. Neural Networks*, vol. 3, pp. 334–338, Mar. 1992.

[13] G. Alefeld and J. Herzberger, *Introduction to Interval Computation*. Reading, MA: Addison-Wesley, 1986.

[14] D. Anguita, S. Ridella, and S. Rovetta, "Circuital implementation of support vector machines," *Electron. Lett.*, vol. 34, no. 16, pp. 1596–1597, 1998.

[15] D. Anguita and A. Boni, "Improved neural network for SVM learning," *IEEE Trans. Neural Networks*, vol. 13, pp. 1243–1244, Sept. 2002.

[16] M. J. Perez-Ilzarbe, "Convergence analysis of a discrete-time recurrent neural network to perform quadratic real optimization with bound constraints," *IEEE Trans. Neural Networks*, vol. 9, pp. 1344–1351, Nov. 1998.

[17] D. Anguita, A. Boni, and S. Ridella, "Learning algorithm for nonlinear support vector machines suited for digital VLSI," *Electron. Lett.*, vol. 35, no. 16, 1999.

[18] S. Fine and K. Scheinberg, "Incremental learning and selective sampling via parametric optimization framework for SVM ," in *Advances in Neural Information Processing Systems 14*, T. G. Dietterich, S. Becker, and Z. Ghahramani, Eds. Cambridge, MA: MIT Press, 2002.

[19] J. Platt, "Fast training of support vector machines using sequential minimal optimization ," in *Advances in Kernel Methods—Support Vector Learning*, B. Schölkopf, C. Burges, and A. Smola, Eds. Cambridge, MA: MIT Press, 1999.

[20] S. S. Keerthi and E. G. Gilbert, "Convergence of a generalized SMO algorithm for SVM classifier design," *Machine Learning*, vol. 46, pp. 351–360, 2002.

[21] J. M. Torres-Moreno and M. B. Gordon, "Characterization of the sonar signals benchmark," *Neural Processing Lett.*, vol. 7, pp. 1–4, 1998.

[22] S. Chen, G. J. Gibson, C. F. N. Cowan, and P. M. Grant, "Adaptive equalization of finite nonlinear channels using multilayer perceptrons, signal processing," *Signal Processing*, vol. 20, no. 2, pp. 107–119, 1990.

[23] D. J. Sebald and J. A. Bucklew, "Support vector machine techniques for nonlinear equalization," *IEEE Trans. Signal Processing*, vol. 48, pp. 3217–3226, Nov. 2000.

[24] *Virtex II Platform FPGA Handbook (ver. 1.3)*, Xilinx.

[25] S. McBader, L. Clementel, A. Sartori, A. Boni, and P. Lee, "Softtotem: An FPGA implementation of the totem parallel processor," presented at the 12th Int. Conf. Field Programmable Logic Application, France, 2002.

Davide Anguita (S'93–M'93) received the Laurea degree in electronic engineering in 1989 and the Ph.D. degree in computer science and electronic engineering from the University of Genova, Genoa, Italy, in 1993.

After working as a Research Associate at the International Computer Science Institute, Berkeley, CA, on special-purpose processors for neurocomputing, he joined the Department of Biophysical and Electronic Engineering, University of Genova, where he teaches digital electronics, programmable logic devices, and smart electronic systems. His current research focuses on industrial applications of artificial neural networks and kernel methods and their implementation on electronic devices.

Andrea Boni received the Laurea degree in electronic engineering in 1996 and the Ph.D. degree in computer science and electronic engineering from the University of Genova, Genoa, Italy, in 2000.

After working as a Research Consultant with the Department of Biophysical and Electronic Engineering, University of Genova, he joined the Department of Information and Communication Technologies, University of Trento, Trento, Italy, where he teaches digital electronics and adaptive electronic systems. His main scientific interests focus on the study and development of digital circuits for advanced information processing.

Sandro Ridella (M'93) received the Laurea degree in electronic engineering from the University of Genova, Genoa, Italy, in 1966.

He is a Full Professor in the Department of Biophysical and Electronic Engineering, University of Genova, Italy, where he teaches inductive learning and statistics and optimization methods. In the last ten years, his scientific activity has been mainly focused in the field of neural networks.