Using Variable Neighborhood Search to Improve the Support Vector Machine Performance in Embedded Automotive Applications

Enrique Alba, Davide Anguita, Member, IEEE, Alessandro Ghio, Student Member, IEEE, Sandro Ridella, Member, IEEE

Abstract—In this work we show that a metaheuristic, the Variable Neighborhood Search (VNS), can be effectively used in order to improve the performance of the hardware-friendly version of the Support Vector Machine (SVM). Our target is the implementation of the feed-forward phase of SVM on resource-limited hardware devices, such as Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs). The proposal has been tested on a machine-vision benchmark dataset for embedded automotive applications, showing considerable performance improvements respect to previously used techniques.

I. INTRODUCTION

Intelligent sensor networks and embedded hardware are getting more and more attention and are used in a large number of applications [1], [2], [3], [4]. These devices are often based on very simple architectures, like Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) and microcontrollers. Such hardware architectures are characterized by hard constraints regarding both power consumption and silicon surface occupation, putting very tightening restrictions to computing power: in many cases, for example, a Floating-Point Unit (FPU) is often unavailable.

The lack of a FPU can be coped by implementing a bit-width reduced FPU [5], [6] or through a reformulation of complex floating-point problems into an integer (or fixed-point) framework: the latter was the case of the backpropagation algorithm for neural network training on resource-limited architectures [7], [8], [9]. Even if the complexity of modern general purpose computers decreased the interest in this kind of applications, the great success of embedded devices in the last years revamped the research in the field of hardware-friendly reformulation of algorithms.

In particular, in this paper we target the hardware implementation of the Support Vector Machine (SVM), considered the state-of-the-art algorithm for classification. Many proposals can be found in the literature on how to cope with the main problems of the hardware implementation of SVM [10], [11]. In several papers [12], [13], some methods are analyzed in order to find a hardware-friendly version of the SVM: an advanced optimization technique allows to find an effective solution, but long computation times are needed [12]; alternatively, it is possible to obtain an approximated solution with shorter computation times, but at the expense of a larger number of bits necessary to reach satisfactory error rates [13]. For this reason, the philosophy of metaheuristic techniques can be helpful in this task, because they can represent a compromise between the two approaches.

Many definitions for metaheuristics can be found in the literature [14], [15], [16]. In short, metaheuristics are high level strategies which can effectively explore the search space in order to find the optimal solution to a problem. Many metaheuristics are available [17], and some techniques can be considered as smart extensions of local search algorithms. This is the case, for example, of the Variable Neighborhood Search (VNS) [18], [19], in which increasingly larger neighborhoods are sampled so to exhaustively explore the solution space.

In this paper, we show how SVM and VNS can be joined effectively: in particular, Sections II and III briefly review, respectively, VNS and the SVM for digital hardware, while Section IV presents the proposed algorithm, in which the two techniques are combined. Finally, Section V shows the experimental results on an automotive application, consisting of pedestrian detection in low-resolution images.

II. VARIABLE NEIGHBORHOOD SEARCH

VNS [18], [19] is a metaheuristic technique which applies a strategy targeted to search for the optimal solution by using a finite set of different neighborhood structures. It is also called a trajectory method, since the algorithm creates a sort of path till the optimal point. The algorithm is very general and many degrees of freedom are present: it should be tuned by the user accordingly to the needs of the problem.

The framework of VNS is shown in Algorithm 1. A set of neighborhood structures \(N_c\), with \(c = 1, \ldots, c_{\text{max}}\), has to be defined by the user. As a first step, an initial solution has to be generated and a neighborhood index is initialized. VNS, then, iterates until one (or more) potential termination condition is met. Each iteration consists in 3 phases: shaking, local search and move. During the shaking phase, a new solution \(s'\) is randomly selected in the \(N_c(s)\) structure of the actual best solution \(s\). \(s'\) is then used for a local search procedure in the second phase of VNS, which allows to find a new solution \(s''\). In the move stage, the final solution \(s''\) is compared against \(s\) based on a fitness function \(f(\cdot)\); if the new solution performs better than the previous best, it is chosen as the new optimum and the index is re-initialized to 1; otherwise, the index
of the neighborhood is incremented (i.e. the neighborhood is enlarged). Obviously, the quality of the final solution is strictly linked to the problem customization and to the quality of the starting solution (i.e. the initial $s$ is not “too far” from the optimum).

Algorithm 1: The VNS algorithm.

Input: A set of neighborhood structures $\mathcal{N}_c$ 
Output: The best solution found 

$s = $ GenerateStartingSolution();

while termination conditions not met do
  $e = 1$;
  while $e < e_{\text{max}}$ do
    $s' = $ PickAtRandom($\mathcal{N}_c(s)$);
    $s'' = $ LocalSearch($s'$);
    if $f(s'') < f(s)$ then
      $s = s''$;
      $e = 1$;
    else
      $e = e + 1$;
  end
end

When facing high dimensional datasets, the local search step can be computationally heavy. Two solutions can be found to this issue: (i) using a “light” local search procedure, or (ii) using the Reduced Variable Neighborhood Search (RVNS) [18], in which the local search phase is removed. The first alternative was used in our proposal since it is often preferable: in fact, removing the local search phase could notably increase the necessary time to meet the termination conditions.

III. THE SUPPORT VECTOR MACHINE FOR DIGITAL HARDWARE

Let us consider a dataset composed by $l$ patterns $\{(x_i, y_i), \ldots, (x_i, y_i)\}$ where $x_i \in \mathbb{R}^m$ and $y_i = \pm 1$. The SVM learning phase consists in solving the following Constrained Quadratic Programming (CQP) problem:

$$
\begin{align*}
\min_{\alpha} & \quad \frac{1}{2} \alpha^T Q \alpha + r^T \alpha \\
\text{subject to} & \quad 0 \leq \alpha_i \leq C \quad \forall i \in [1, \ldots, l]
\end{align*}
$$

where $r_i = -1 \forall i$, and $Q$ is a symmetric positive semidefinite $l \times l$ matrix $q_{ij} = y_i y_j K(x_i, x_j)$ defined through a Mercer kernel $K(\cdot, \cdot)$ [20]. After solving the above problem, the feed-forward phase can be computed as

$$
f(x) = \sum_{i=1}^{l} y_i \alpha_i K(x_i, x).
$$

Note that, differently from the conventional CQP problem [20], we omit the equality constraint $\sum_i y_i \alpha_i = 0$, which is equivalent to search for a SVM without the bias term ($b = 0$). The advantage of this simpler formulation is obvious, when dealing with limited precision variables, because it allows us to avoid the computation of $b$. In fact, it is well–known that the bias term does not appear in Eq. (1) and it must be computed using the Karush–Kuhn–Tucker (KKT) conditions [21], which are satisfied only at the solution point $\alpha^*$. Since our objective is to find a fixed–point solution, which will differ, in general, from $\alpha^*$, it would be incorrect to compute the bias term through the KKT conditions [22]. It can be noted, however, that the theoretical framework of the SVM is preserved, even with $b = 0$, if the kernel satisfies some weak density properties [23]. Valid kernels are, for example, the Gaussian and the Laplacian ones: in these cases, the bias term can be safely removed.

In order to build a hardware–friendly SVM, we perform the following normalization

$$
\beta_i = \frac{\alpha_i 2^n - 1}{C}
$$

and apply it to Eq. (1), obtaining

$$
\begin{align*}
\min_{\beta} & \quad \frac{1}{2} \beta^T Q \beta + s^T \beta \\
\text{subject to} & \quad 0 \leq \beta_i \leq 2^n - 1 \quad \forall i \in [1, \ldots, l]
\end{align*}
$$

where $s_i = -\frac{2^n - 1}{C}$ \forall $i$. Note that this formulation provides the same solution of the original one, but allows us to remove the regularization constant $C$ from the constraints, making it possible to represent the integer part of the coefficients of the SVM with exactly $n$ bits. Usually, the training phase for finding the optimal coefficients is performed offline on a general purpose PC.

The feedforward phase of the SVM can now be performed online on a resource–limited device and is computed by

$$
f(x) = \sum_{i=1}^{l} y_i \beta_i K(x_i, x).
$$

The output of the SVM is scaled by $2^{p-1}$, respect to the original one, but its sign, which corresponds to the classification label, is not affected.

Any dense positive definite kernel can be used in the above formulation, but a Laplacian kernel is easier to implement in digital hardware, since it avoids the use of expensive multipliers and requires only some shifters and adders [24], [25]. We choose here the following Laplacian kernel:

$$
K(x_i, x) = 2^{-\gamma \|x_i - x\|_1}
$$

where $\gamma > 0$ is the kernel hyperparameter and the 1-norm is defined as $\|x\|_1 = \sum_i |x_i|$.

The problem of Eq. (4) can be solved with an integer solution by rounding the corresponding floating–point solution to the nearest integer value $\beta^{NI}$ [13]. This method, called Nearest Integer SVM (NI–SVM), is summarized in Algorithm 2. Even if with NI–SVM it is possible to obtain immediately an integer solution, it does not perform as well as vectors found using a branch–and–bound technique in terms of error rate. Anyhow, the branch–and–bound often needs more than $10^9$ seconds for finding the final solution on large (i.e., more than 1000 training patterns) datasets [12].
Algorithm 2: The NI–SVM algorithm.

Input: A training set \{\((x_1, y_1), \ldots, (x_l, y_l)\)\} where \(x_i \in \mathbb{R}^m\) and \(y_i = \pm 1\)
Output: The nearest integer solution \(\beta^{NI}\)
\[ \beta = \text{SVM}; \]
\[ \beta^{NI} = \text{RoundToNearest}(\beta); \]

Algorithm 3: The SVM–VNS algorithm. In Algorithm 4, the CreateNeighbor(\(\cdot,\cdot\)) function is presented.

Input: A training set \{\((x_1, y_1), \ldots, (x_l, y_l)\)\} where \(x_i \in \mathbb{R}^m\) and \(y_i = \pm 1\)
Output: The best solution found \(\beta^{VNS}\)
\[ \beta^{VNS} = \text{NI–SVM}; \]
\[ s = \beta^{NI}; \]
\[ e = 1; \]
while \(e < l\) do
  for \(i = 1\) to \(n_n\) do
    \(s'[i] = \text{CreateNeighbor}(s,e);\)
  end
  \(s'' = \text{BestSolution}(s');\)
  if \(f(s'') < f(s)\) then
    \(s = s'';\)
    \(e = 1;\)
  else
    \(e = e + 1;\)
  end
end
\[ \beta^{VNS} = s \]

IV. THE COMBINED SVM–VNS APPROACH

The NI–SVM method generates a sub–optimal solution \(\beta^{NI}\) but in a very short time and, at the same time, the VNS algorithm needs a good starting point in order to reach better performance. Therefore, it is possible to use \(\beta^{NI}\) as a starting point for our customized version of VNS. The SVM–VNS approach is obtained and presented in Algorithm 3.

Algorithm 4: The CreateNeighbor(\(\cdot,\cdot\)) function of Algorithm 3.

Input: The number of components to be perturbed \(e\) and the actual solution \(s\)
Output: The perturbed vector \(s_p\)
\[ s_p = s; \]
\[ \text{for } j = 1 \to e \text{ do } \]
\[ i = \text{UniformRNG}(); \]
\[ s_p(i) = \mathcal{P}_{0,2^{n-1}}[\text{round}(s(i) + NRNG(0,n))]; \]
end

V. EXPERIMENTAL RESULTS

We tested our method on an automotive machine vision application, where the use of embedded systems is widespread. The problem consists in detecting pedestrians in an image, making use of the DaimlerChrysler dataset [26], which has been introduced to benchmark people detection systems. The dataset consists of 8–bits grayscale images (18×36 pixels), divided in 4900 patterns representing a pedestrian crossing a road and 4900 non–pedestrian examples. Figure 1 shows examples of one image containing a pedestrian (on the left) and one representing a road sign (a non–pedestrian example, on the right). Because of the possible instability of the results (due to the adopted metaheuristic), preventing from highlighting the absolute quality of the method, we use eight randomly resampled replicates of the entire dataset, each one split between a training (1225 samples) and a test set (8575 patterns).

As a matter of fact, in our case, the neighborhood structures \(\mathcal{N}_e\), with \(e = 1, \ldots, l\), contain all the possible realizations of the solution vector, where from 1 to \(l\) components are perturbed. The local search stage consists in finding the \(s'\) vector which performs best according to the fitness function. Finally, the move phase consists in selecting the best solution so far. After the algorithm ends, the final best solution \(s\) is assigned to the output vector \(\beta^{VNS}\), which can be used for the feed–forward phase of the SVM, Eq. (6).

The first step of the algorithm consists in finding the nearest integer solution of the SVM \(\beta^{NI}\). The fitness function \(f(\cdot)\) is chosen to be the value of the dual cost of Eq. (4) in order to maintain the framework of the original CQP problem.

At the beginning of the run, during the shaking phase, VNS finds \(n_n\) neighbors, using the function described in Algorithm 4: \(e\) components of the solution, randomly chosen by a Random Number Generator (RNG) with a uniform distribution, are perturbed with a Gaussian random noise. The latter is characterized by 0 mean and variance equal to \(n\) and is generated with a Normal RNG (NRNG). Then, the modified components are rounded to the nearest integer value and re–projected (if necessary) into the solution bounds accordingly to:

\[ \mathcal{P}_{a,b}[c] = \begin{cases} a & \text{if } c < a \\ c & \text{if } a \leq c \leq b \\ b & \text{if } c > b. \end{cases} \] (8)
memory, running Microsoft Windows 2003 Server 64 bit Edition and the Intel 9.1 Fortran compiler. Obviously, the code could benefit from the multi-core architecture, but this is not the target of our analysis. Therefore, the code has been compiled so to make use of a single core.

The experimental setup is the following for each replicate of the dataset:

- both training and test data are normalized in the range [0, 1];
- the SVM–VNS is performed, using \( n_n \) neighbors for each step of VNS;
- the optimal 64 bits floating–point (FP) solution \( \beta \), the NI–SVM vector \( \beta^{NI} \) as well as the final one \( \beta^{VNS} \) are saved. The corresponding values of the elapsed times for calculating the three solutions are stored as well;
- the performance of \( \beta \), \( \beta^{NI} \) and \( \beta^{VNS} \) are evaluated by classifying the dataset using the feed–forward function of Eq. (6). An output equal to 0 has been considered as a misclassification.

For each experiment, 30 replicates using different seed values for the RNGs of the SVM–VNS algorithm are performed. This allows to compute the associated \( p \)-value to decide the statistical quality of the results by a Student’s \( t \)-test.

The optimal hyperparameter values for each dataset are found using a grid search, in order to test several pairs \((C, \gamma)\), and a 10–fold Cross–Validation procedure to estimate the generalization error. Alternatively, some more advanced techniques could be used in order to find the optimal hyperparameter values [27], [28], [29], [30], [31]. The average error rate for the floating–point solution \( \beta \) is 3.10\%, with a standard deviation equal to 0.45. Table I shows the average misclassification error and the corresponding standard deviation computed on the eight replicates, varying the number of bits used for the solutions as well as the number of neighbors per step \( n_n \) in SVM–VNS. The columns report the results obtained by varying the number of bits \( n \). The first row refers to the nearest–integer solution \( \beta^{NI} \), found using the NI–SVM approach; the following rows refer to the combined SVM–VNS approach.

An integer solution is considered to perform comparably to the floating–point one if the corresponding \( t \)-test is passed with a confidence equal or greater than 95\%. The best solutions for each method, found row–by–row, are marked in Table I using bold face. As it can be seen, the solution given by the NI–SVM \( \beta^{NI} \) does not deteriorate even when using only 12 bits. The new method allows to obtain the same performance with 8 bits. Furthermore, even the solution with 4 bits is practically (though not statistically) comparable to the floating–point one.

Table II presents the average elapsed time, necessary for finding the final solutions. By comparing Table II with Table I, it is possible to see that, in a limited computation time, it is possible to improve the quality of the final solution. For example, setting \( n = 4 \) bits, in about 2500 seconds a solution which performs similarly to the floating–point one is found, characterized by more than the 30\% improvement respect to \( \beta^{NI} \). It is interesting to see that the mean elapsed times, independently from the number of bits, are almost constant for each \( n_n \) value and grow linearly with the number of neighbors per step. A typical trend is shown in Figure 2. Then, the necessary time for a general number of neighbors \( n_n \) can be approximated by:

\[
time \approx 25 \cdot n_n + time_{NI} \quad (9)
\]

where \( time_{NI} \) is the elapsed time for calculating the NI–SVM solution (in our case, \( time_{NI} \approx 2 \) seconds).

VI. CONCLUSIONS

We proposed a new method for improving the Support Vector Machine performance on resource–limited hardware for automotive applications. The Variable Neighborhood Search is used because it allows to find solutions which are a good compromise between the performance and the needed computation time. The experimental results show that it is possible to implement a SVM with a number of bits which is much lower than previous methods appeared in the literature.

ACKNOWLEDGMENT

The first author acknowledges funding by the Spanish Ministry of Education and Science and FEDER under contract TIN2005–08818–C04–01 (the OPLINK project). Part of this work has been realized in the framework of the

### TABLE I

<table>
<thead>
<tr>
<th>( n_n = 16 )</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI–SVM</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.11 \pm 0.46 )</td>
<td>( 3.30 \pm 0.63 )</td>
<td>( 34.53 \pm 17.36 )</td>
</tr>
<tr>
<td>SVM–VNS</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.11 \pm 0.42 )</td>
<td>( 3.60 \pm 0.50 )</td>
</tr>
<tr>
<td>SVM–VNS</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.11 \pm 0.42 )</td>
<td>( 3.46 \pm 0.52 )</td>
</tr>
<tr>
<td>SVM–VNS</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.10 \pm 0.45 )</td>
<td>( 3.11 \pm 0.42 )</td>
<td>( 3.39 \pm 0.50 )</td>
</tr>
</tbody>
</table>

### TABLE II

<table>
<thead>
<tr>
<th>( n_n = 16 )</th>
<th>12</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>NI–SVM</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SVM–VNS</td>
<td>775</td>
<td>754</td>
<td>753</td>
<td>760</td>
</tr>
<tr>
<td>SVM–VNS</td>
<td>1267</td>
<td>1264</td>
<td>1263</td>
<td>1270</td>
</tr>
<tr>
<td>SVM–VNS</td>
<td>2498</td>
<td>2524</td>
<td>2521</td>
<td>2550</td>
</tr>
</tbody>
</table>

REFERENCES


[27] B.L. Milenova, J.S. Yarmus, M.M. Campos, “SVM in Oracle Database 10g: Removing the Barriers to Widespread Adoption of Support Vector Machines”, in Proc. of the 31st Int. Conf. on Very Large Data Bases, Trondheim, Norway, pp. 1152–1163, Aug. 30 - Sep. 02, 2005.


