AUTOMATIC FPGA BASED IMPLEMENTATION OF A CLASSIFICATION TREE

J. Mitéran[1], J. Matas[2], J. Dubois[1], E. Bourennane[1]
[1] Le2i - FRE CNRS 2309 Aile des Sciences de l'ingénieur
Université de Bourgogne - BP 47870 - 21078 Dijon - FRANCE
miteranj@u-bourgogne.fr
[2] Center of Machine Perception – CVUT, Karlovo Namesti 13, Prague, Czech Republic

Abstract

We propose a method of automatic hardware implementation of a decision rule based on the Adaboost algorithm. We review the principles of the classification method and we evaluate its hardware implementation cost in term of FPGA's slice, using different weak classifiers based on the general concept of hyperrectangle. We show how to combine the weak classifiers in order to find a good trade-off between classification performances and hardware implementation cost. We present results obtained using examples coming from UCI databases.

Keywords : Adaboost, FPGA, classification, hardware, image segmentation

1 INTRODUCTION

In this paper, we propose a method of automatic hardware implementation of a particular decision rule. This paper focuses mainly high speed decisions (approximately 5 to 10 ns per decision) which can be useful for hi-resolution image segmentation or pattern recognition tasks in very large image databases. Our work is designed in order to be easily integrated in a System-On-Chip, which can perform the full process: acquisition, feature extraction and classification. This paper focuses on the last part of this process. Our method is based on the well known Adaboost algorithm, which decision consists in a simple summation of signed numbers [1, 2]. The limited number of operations to be performed allows us to choose the fastest implementation, a fully parallel one. Moreover, the regular structure of the function can be automatically generated using a hardware description language such as VHDL, and thus can be implemented efficiently in FPGA.

Many implementations of particular classifiers have been proposed, mainly based on neural networks [3, 4, 5]. However, the implementation of a classifier is not often optimum in terms of silicon area and performances, because of the general structure of the chosen algorithm. Moreover Adaboost is a powerful machine learning method that can be applied directly, without any modification to generate a classifier implementable in hardware, and a complexity/performance trade-off is natural in the framework: Adaboost learning constructs a set of classifier with increasing complexity and better performance (lower crossvalidated error).

In order to follow real-time processing and cost constraint, we have to minimise the test error e while minimising the hardware implementation cost λ and maximise the decision speed. The maximum speed will be obtained using a full parallel implementation. We estimated λ considering Field Programmable Gate Array (FPGA) as the hardware target. The advantage of these components is mainly their reconfigurability [6] [7]. Using reconfigurable architecture, it is possible to integrate the constant values in the design of the decision function, optimising the number of cells used. We consider here the slice as the main elementary structure of the FPGA and the unit of λ. One component can contain a few thousand of these blocks.

In the first part of this paper, we present the principle of the proposed method, reviewing the Adaboost algorithm and defining a family of weak classifiers suitable to hardware implementation, based on the general concept of hyperrectangle. We describe how it is possible to estimate the full parallel hardware implementation cost in terms of slices. In the second part, we present the algorithm allowing finding a hyperrectangle minimizing the classification error and allowing finding a good trade-off between performance hardware implementation cost which we estimated. In the third part, results obtained on real databases coming from UCI repository are presented.

2 PROPOSED METHOD

2.1 Review of Adaboost

The basic idea introduced by Schapire and Freund [1, 2] is that a combination of single rules or "weak classifiers" gives a "strong classifier". Each sample is defined by a feature vector $\mathbf{x}=(x_1, x_2, ..., x_D)^T$ in an D dimensional space and its corresponding class :

$C(\mathbf{x}) = y \in \{-1,+1\}$ in the binary case.

We define the learning set S of p samples as:

$$S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_p, y_p)\}.$$

Each sample is weighted such as after each iteration of the process (which consists in finding the best weak classifier as possible), the weights w_i of the misclassified samples are increased, and the weights of the well classified sample are decreased. The final class y is given by:

$$y(\mathbf{x}) = sgn\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})\right)$$

Where both α_t and h_t are to be learned by the following boosting procedure.

1. Input $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_p, y_p)\}$ *, number of iteration T and initialize* $w_i^{(t)} = 1/p$ *for all i=1, ..., p*

2. Do for t=1, ..., T

2.1 Train classifier with respect to the weighted samples set $\{S, \mathbf{d}^{(t)}\}$ *and obtain hypothesis* $h_t : x \to \{-1, +1\}$

2.2 Calculate the weighted error ε_t *of* $h_t : \varepsilon_t = \sum_{i=1}^{p} d_i^{(t)} \mathrm{I}(y_i \neq h_t(\mathbf{x}_i))$

2.3 Compute the coefficient $\alpha_t = \frac{1}{2} \log \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$

2.4 Update the weights $d_i^{(t+1)} = \frac{d_i^{(t)}}{Z_t} \exp\{-\alpha_t y_i h_t(\mathbf{x}_i)\}$

Where Zt is a normalization constant: $Z_t = 2\sqrt{\varepsilon_t (1 - \varepsilon_t)}$

3. Stop if $\varepsilon_t = 0$ *or* $\varepsilon_t \geq \frac{1}{2}$ *and set T=t-1*

4. Output : $y(\mathbf{x}) = sgn\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \right)$

2.2 Choice of a good weak classifier

A weak classifier suitable to parallel hardware implementation is necessary. In term of slices, the hardware cost can be expressed as follow:

$$\lambda = (T-1)\lambda_{add} + \lambda_T$$

where λ_{add} is the cost of an adder (which will be considered as a constant here), and λ_T is the cost of the parallel implementation of the set of the weak classifiers :

$$\lambda_T = \sum_{t=1}^{T} \lambda_t$$

where λ_t is the cost of the weak classifier h_t associated to the multiplexers (see Fig. 1).

Single parallel axis threshold is often used in the literature. However, the number of iterations needed by a so simple classifier is often important, increasing the hardware cost (which depends on the number of additions to be performed in parallel). To increase the complexity of the weak classifier allows converging faster, and then minimizing the number of additions, but will also increase the second member of the equation. We have then to find a trade off between the complexity of h_t and the hardware cost.

It has been proved in the literature that decision trees based on hyperrectangles (or union of boxes) instead of single threshold give better results [11]. Moreover, the decision function associated with a hyperrectangle can be easily implemented in parallel (Fig. 2).

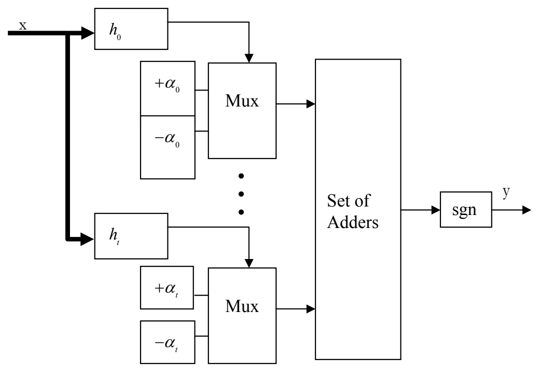

Fig. 1 Parallel implementation of Adaboost

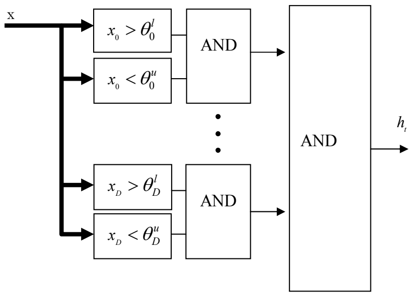

Fig. 2 Parallel implementation of h_t

However, there is no algorithm in the complexity of D allowing finding the best hyperrectangle, i.e. minimizing the learning error. We will use a suboptimum algorithm to find it.

We defined the hyperrectangle as a set H of $2D$ thresholds and a class y_H

$$H = \{\theta_1^l, \theta_1^u, \theta_2^l, \theta_2^u, \dots, \theta_D^l, \theta_D^u, y_H\}$$

Where θ_k^l and θ_k^u are respectively the lower and upper limits of a given interval in the kth dimension. The decision function is

$$h_H(\mathbf{x}) = y_H \Leftrightarrow \prod_{d=1}^{D} (x_d > \theta_d^l)(x_d < \theta_d^u) \text{ , } h_H(\mathbf{x}) = -y_H \text{ otherwise}$$

This expression, where product is the logical operator, can be simplified if some of these limits are rejected to the infinite (or 0 and 255 in case of byte based implementation). Comparisons are not necessary in this case since the result will be always true. It is particularly important for minimising the final number of used slices. Two particular cases have to be considered:

The single threshold: $\Gamma = \{\theta_d, y_\Gamma\}$

Where θ_d is a single threshold, $d \in \{1, \dots, D\}$, and the decision function is:

$$h_\Gamma(\mathbf{x}) = y_\Gamma \Leftrightarrow x_d < \theta_d \text{ , } h_\Gamma(\mathbf{x}) = -y_\Gamma \text{ otherwise}$$

The single interval: $\Delta = \{\theta_d^l, \theta_d^u, y_\Delta\}$

Where the decision function is:

$$h_\Delta(\mathbf{x}) = y_\Delta \Leftrightarrow (x_d > \theta_d^l) \text{and} (x_d < \theta_d^u) \text{ , } h_\Delta(\mathbf{x}) = -y_\Delta \text{ otherwise}$$

In these two particular cases, it is easy to find the optimum hyperrectangle, because each feature is considered independently form the others. In the general case, one has to follow a particular heuristic given a suboptimum hyperrectangle. A family of such classifiers have been defined, based on the NGE algorithm described by Salzberg [12] whose performance was compared to the Knn method by Wettschereck and Dietterich [13]. This method divides the attribute space into a set of hyperrectangles based on samples. The performance of our own implementation was studied in [14]. We will review the principle of the hyperrectangle determination in the next paragraph.

3 HYPERRECTANGLE DETEMINATION

3.1 Review of Hyperrectangle based method

The core of the strategy is the hyperrectangles set H determination from a set of sample S.

During the first step, one hyperrectangle $H(\mathbf{x})$ is build for each sample $\mathbf{x}$ of the learning set S as follows: each part Q_k (see **Fig. 3**) defines the area where for all sample $\mathbf{u} \in Q_k, \mathrm{d}_\infty(\mathbf{x},\mathbf{u}) = |x_k - u_k|$ with:

$$d_\infty(\mathbf{u},\mathbf{v}) = \max_{k=1,\dots,D} |u_k - v_k|$$

We determine $\mathbf{z}$ as the nearest neighbour belonging to a different class in each part Q_k. If d_k is the distance between $\mathbf{x}$ and $\mathbf{z}$ in a given Q_k, the limit of the hyperrectangle is computed as $d_f = d_k R$. The parameter R should be less or equal to 0.5. This constraint ensures that the hyperrectangle cannot contain any sample of opposite classes.

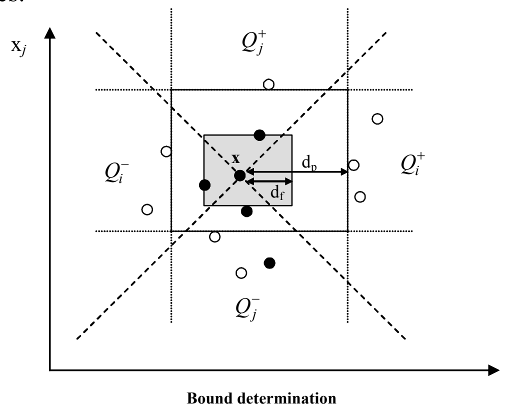

Fig. 3 Hyperrectangle computation

During the second step, hyperrectangles of a given class are merged together in order to eliminate redundancy (hyperrectangles which are inside of other hyperrectangle of the same class). We obtain a set H of hyperrectangles :

$$\mathrm{H} = \{H_1, H_2, \dots, H_q\}$$

We evaluated the performance of this algorithm in various cases, using theoretical distributions as well as real sampling [8]. We compared the performance with neural networks, the Knn method and a Parzen's kernel based method [10]. It clearly appears that the algorithm performs poorly when the inter-class distances are too small: an important number of hyperrectangles are created in the overlap area, slowing down the decision or increasing the implementation cost. However, it is possible to use the hyperrectangle generated as a step of the Adaboost process, selecting the best one in terms of classification error.

3.2 Boosting general Hyperrectangle

From H we have to build one hyperrectangle minimising the weighted error. To obtain this result, we merge hyperrectangles following a one-to-one strategy, thus building $q'=q(q-1)$ new hyperrectangles. We keep H_{opt} the hyperrectangle giving the smallest weighted error.

3.3 Estimation of the hyperrectangle hardware implementation cost

It is possible to estimate the hardware implementation cost of h_t, taking into account that we can code the constant values of the decision function into the final architecture, using the advantage of FPGA based reconfigurable computing. Indeed, the binary result L_B of the comparison of the variable byte A and the constant byte B is a function F_B of the bits of A:

$$L_B = F_B(A_7, A_6, \dots, A_0)$$

In the worst case, the particular structure of L_B can be stored in two cascaded Look Up Tables (LUT) of 16 bits each (one slice). We have coded a tool which automatically generates a VHDL description of a decision function given the result of a training step (i.e. given the hyperrectangles limits). We then have used a standard synthesizer tool for the final implementation in FPGA.

In the case of single threshold, $\lambda_t = 1$. In the case of interval, $\lambda_t \le 2$. In the case of general hyperrectangle, the decision rule requires in the worst case 2 comparators per hyperrectangle and per feature: $\lambda_t \le 2D$. Considering that some limits of the general hyperrectangle can be rejected to the "infinit", the general cost can be expressed as follows:

$$\lambda \le (T-1)\lambda_{add} + kT \ , \ k \le 2D$$

where k is the number of lower limits of hyperrectangles which are greater than 0 plus the number of upper limits which are lower than 1 (or 255 in the byte case).

4 RESULTS

We applied our method in different cases, based on real databases coming from UCI repository. These example are more significant in terms of hardware implementation, since they are performed in high dimensional spaces (until D=64, this can be seen as a reasonable limit for a full parallel implementation).

For each example and, we give also the result of a decision based on SVM developed by Vladimir Vapnik [8], which is known as one of the best classifier, and which can be compared with Adaboost on the theoretical point of view. At the same time SVM can achieve good performance when applied to real problems [15, 16, 17,18]. In order to compare the implementation cost of the two methods, we evaluated the hardware implementation cost of SVM as:

$$\lambda_{svm} \simeq 72(3D-1)Ns + 8$$

Where *Ns* is the total number of “Support Vectors” determined during the training step. We used here a RBF based kernel, using distance L1. While the decision function seems to be similar to the Adaboost one’s, the cost is here mainly higher because of multiplications, even if the exponential function can be stored in a particular look up table (LUT) to avoid computation, the kernel product K requires some multiplications and additions; the final decision function requires at least one multiplication and one addition per support vector

$$\mathrm{C}(\mathbf{x}) = \mathrm{Sgn}\left(\sum_{i=1}^{Ns} y_i \alpha_i \cdot K(\mathbf{s}_i, \mathbf{x}) + b\right)$$

Results are summarised in the Table 1. The number of classes *c* is from 2 to 10. For each case, we give the result of classification using a RBF kernel based SVM as a reference. One can see that the direct hardware cost of this classifier is not realistic here. Considering the different results of our Adaboost implementation, it appears clearly that the combination of the three types of weak classifiers gives the better results. The optdigit and the pendigit cases can be solved using from 2 to 5 component of the Virtex family, for example, while all the other cases can be implemented in a single low cost chip. Moreover, the classification error of the Adaboost based classifier is very close to the SVM one.

Table 1 Results on real databases

Database	D	c	SVM (RBF)		Threshold		Interval		Hyperrectangle	
			e (%)	λ_{SVM}	e (%)	λ	e (%)	λ	e (%)	λ
optdigit	64	10	1.15	20215448	2.605	5292.5	2.735	5414	2.59	4392.5
pendigit	16	10	0.625	2270672	20.875	3435	2.01	5481.5	1.415	3405.5
Ionosphere	34	2	7.95	465416	8.23	126	6.81	149.5	7.095	119.5
IMAGE	17	7	3.02	1699208	12.91	568.5	7.655	697	4.015	973.5
WINE	13	3	4.44	87560	3.33	98	5.525	98	6.11	18

5 CONCLUSION

We have developed a method allowing automatic generation of hardware implantation of a particular decision rule based on the Adaboost algorithm, which can be applied in many pattern recognition tasks, such as pixel wise image segmentation, character recognition, etc.

We experimentally validated the method on real cases, coming from standard datasets. We demonstrated that it is possible to find a good trade off between the hardware implementation cost and the classification error. The final error of this classifier if often very close to the SVM error, which can be seen as a good reference. Moreover, the method is really easy to use, since the only parameters to tune are the choice of the weak classifier and the R value of the hyperrectangle based method. We are currently finalizing a development tool which will allows developing the whole implementation process, from the learning set definition to FPGA based implementation using automatic VHDL generation. Our future work will be the integration of this method as a standard IP generation tool for classification.

6 REFERENCES

[1] R.E. Schapire, The strengh of weak leanabilty Machine Learning, 5(2), pp 197-227, 1990.

[2] Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences, 55(1), pp 119-139, 1997.

[3] C. M. Bishop *Neural networks for Pattern Recognition*, Oxford University Press, 1995, pp 110-230.

[4] P. Lysaght, J. Stockwood, J. Law and D. Girma, Artificial Neural Network, Implementations on a Fine-Grained FPGA, in Field Programmable Logic and Applications, R. Hartenstein, M. Z. Servit (Eds.), Prague, pp 421–431, 1994

[5] Y. Taright, M. Hubin, FPGA Implementation of a Multilayer Perceptron Neural Network using VHDL, 4th Int.l Conf. on Signal Processing (ICSP'98), Beijing,Vol 2, pp 1311-1314, 1998.

[6] R. Enzler, T. Jeger, D. Cottet, and G. Tröster High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs, In *Field-Programmable Logic and Applications* (Proc. FPL 00), Lecture Notes in Computer Science, Vol. 1896, Springer, pp. 525-534, 2000.

[7] S. Hauck, The Roles of FPGAs in Reprogrammable Systems, Proceedings of the IEEE, 86(4): pp 615-638, 1998.

[8] J. Mitéran, P. Gorria and M. Robert, Classification géométrique par polytopes de contraintes. Performances et intégration , *Traitement du Signal*, Vol 11, pp 393-408,1994.

[9] M. Robert, P. Gorria, J. Mitéran, S. Turgis Architectures for real-time classification processor, *Custom Integrated Circuit Conference*, San Diego CA, pp 197-200, 1994.

[10] R. O. Duda and P.E. Hart, *Pattern classification and scene analysis*, Wiley, New York, 1973, pp. 230-243.

[11] I. De Macq, L. Simar, Hyper-rectangular space partitionning trees, a few insight, discussion paper 1024, Université Catholique de Louvain, 2002.

[12] S. Salzberg, A nearest hyperrectangle learning method. *Machine Learning*, 6: pp 251-276, 1991.

[13] D. Wettschereck and T. Dietterich, An Experimental Comparison of the Nearest-Neighbor and Nearest-Hyperrectangle Algorithms, *Machine Learning,* Vol 19, n°1, pp 5-27, 1995.

[14] J. Mitéran, J. P. Zimmer, F. Yang, M. Paindavoine Access control : adaptation and real-time implantation of a face recognition method, *Optical Engineering*, 40(4): pp 586-593, 2001.

[15] V. Vapnik *The nature of statistical learning theory* , Springer-Verlag, New York, 1995.

[16] B. Schölkopf, A. Smola, K.-R. Müller, C. J. C. Burges and V. Vapnik Support Vector methods in learning and feature extraction, *Australian Journal of Intelligent Information Processing Systems,* 1: pp 3-9, 1998.

[17] K. Jonsson, J. Kittler, Y. P. Li, and J. Matas Support Vector Machines for Face Authentication. In T. Pridmore and D. Elliman, editors, British Machine Vision Conference, pp 543-553, 1999.

[18] M. A. Hearst, B. Schölkopf, S. Dumais, E. Osuna and J. Platt, Trends and Controversies - Support Vector Machines. *IEEE Intelligent Systems*, 13(4) : pp 18-28, 1998.