Evolutionary Product-Unit Neural Networks for Classification*

F.J. Martínez-Estudillo[1,**], C. Hervás-Martínez[2], P.A. Gutiérrez Peña[2]
A.C. Martínez-Estudillo[1], and S. Ventura-Soto[2]

[1] Department of Management and Quantitative Methods, ETEA,
Escritor Castilla Aguayo 4, 14005, Córdoba, Spain
Tel.: +34957222120; Fax: +34957222107
fjmestud@etea.com, acme@etea.com
[2] Department of Computing and Numerical Analysis of the
University of Córdoba, Campus de Rabanales, 14071, Córdoba, Spain
chervas@uco.es, zamarck@yahoo.es, sventura@uco.es

Abstract. We propose a classification method based on a special class of feed-forward neural network, namely product-unit neural networks. They are based on multiplicative nodes instead of additive ones, where the nonlinear basis functions express the possible strong interactions between variables. We apply an evolutionary algorithm to determine the basic structure of the product-unit model and to estimate the coefficients of the model. We use softmax transformation as the decision rule and the cross-entropy error function because of its probabilistic interpretation. The empirical results over four benchmark data sets show that the proposed model is very promising in terms of classification accuracy and the complexity of the classifier, yielding a state-of-the-art performance.

Keywords: Classification; Product-Unit; Evolutionary Neural Networks.

1 Introduction

The simplest method for classification provides the class level given its observation via linear functions in the predictor variables. Frequently, in a real-problem of classification, we cannot make the stringent assumption of additive and purely linear effects of the variables. A traditional technique to overcome these difficulties is augmenting/replacing the input vector with new variables, the basis functions, which are transformations of the input variables, and then to using linear models in this new space of derived input features. Once the number and the structure of the basis functions have been determined, the models are linear in these new variables and the fitting is a well known standard procedure. Methods like sigmoidal feed-forward neural networks, projection pursuit learning, generalized additive models [1], and PolyMARS [2], a hybrid of multivariate adaptive splines (MARS) specifically

* This work has been financed in part by TIN 2005-08386-C05-02 projects of the Spanish Inter-Ministerial Commission of Science and Technology (MICYT) and FEDER funds.
** Corresponding author.

E. Corchado et al. (Eds.): IDEAL 2006, LNCS 4224, pp. 1320 – 1328, 2006.

designed for classification problems, can be seen as different basis function models. The major drawback of these approaches is to state the optimal number and the typology of corresponding basis functions. We tackle this problem proposing a nonlinear model along with an evolutionary algorithm that finds the optimal structure of the model and estimates the corresponding parameters. Concretely, our approach tries to overcome the nonlinear effects of variables by means of a model based on nonlinear basis functions constructed with the product of the inputs raised to arbitrary powers. These basis functions express the possible strong interactions between the variables, where the exponents may even take on real values and are suitable for automatic adjustment. The proposed model corresponds to a special class of feed-forward neural network, namely product-unit neural networks, PUNN, introduced by Durbin and Rumelhart [3]. They are an alternative to sigmoidal neural networks and are based on multiplicative nodes instead of additive ones. Up to now, PUNN have been used mainly to solve regression problems [4], [5].

Evolutionary artificial neural networks (EANNs) have been a key research area in the past decade providing a better platform for optimizing both the weights and the architecture of the network simultaneously. The problem of finding a suitable architecture and the corresponding weights of the network is a very complex task (for a very interesting review on this subject the reader can consult [6]). This problem, together with the complexity of the error surface associated with a product-unit neural network, justifies the use of an evolutionary algorithm to design the structure and training of the weights. The evolutionary process determines the number of basis functions of the model, associated coefficients and corresponding exponents. In our evolutionary algorithm we encourage parsimony in evolved networks by attempting different mutations sequentially. Our experimental results show that evolving parsimonious networks by sequentially applying different mutations is an alternative to the use of a regularization term in the fitness function to penalize large networks. We use the softmax activation function and the cross-entropy error function. From a statistical point of view, the approach can be seen as a nonlinear multinomial logistic regression, where we optimize the log-likelihood using evolutionary computation. Really, we attempt to estimate conditional class probabilities using a multilogistic model, where the nonlinear model is given by a product-unit neural network.

We evaluate the performance of our methodology on four data sets taken from the UCI repository [7]. Empirical results show that the proposed method performs well compared to several learning classification techniques. This paper is organized as follows: Section 2 is dedicated to a description of product-unit based neural networks; Section 3 describes the evolution of product-unit neural networks; Section 4 explains the experiments carried out; and finally, Section 5 shows the conclusions of our work.

2 Product-Unit Neural Networks

In this section we present the family of product-unit basis functions used in the classification process and its representation by means of a neural network structure. This class of multiplicative neural networks comprises such types as sigma-pi

networks and product unit networks. A multiplicative node is given by $y_j = \prod_{i=1}^{k} x_i^{w_{ji}}$, where k is the number of the inputs. If the exponents are $\{0,1\}$ we obtain a higher-order unit, also known by the name of sigma-pi unit. In contrast to the sigma-pi unit, in the product-unit the exponents are not fixed and may even take real values.

Some advantages of product-unit based neural networks are increased information capacity and the ability to form higher-order combinations of the inputs. Besides that, it is possible to obtain upper bounds of the VC dimension of product-unit neural networks similar to those obtained for sigmoidal neural networks [8]. Moreover, it is a straightforward consequence of the Stone-Weierstrass Theorem to prove that product-unit neural networks are universal approximators, (observe that polynomial functions in several variables are a subset of product-unit models).

Despite these advantages, product-unit based networks have a major drawback: they have more local minima and more probability of becoming trapped in them [9]. The main reason for this difficulty is that small changes in the exponents can cause large changes in the total error surface and therefore their training is more difficult than the training of standard sigmoidal based networks. Several efforts have been made to carry out learning methods for product units [9],[10]. Studies carried out on PUNNs have not tackled the problem of the simultaneously design of the structure and weights in this kind of neural network, either using classic or evolutionary based methods. Moreover, so far, product units have been applied mainly to solve regression problems. We consider a product-unit neural network with the following structure (Fig. 1): an input layer with k nodes, a node for every input variable, a hidden layer with m nodes and an output layer with J nodes, one for each class level. There are no connections between the nodes of a layer and none between the input and output layers either. The activation function of the j-th node in the hidden layer is given by $B_j(\mathbf{x}, \mathbf{w}_j) = \prod_{i=1}^{k} x_i^{w_{ji}}$, where w_{ji} is the weight of the connection between input node i and hidden node j and $\mathbf{w}_j = (w_{j1}, ..., w_{jk})$ the weights vector. The activation function of each output node is given by $f_l(\mathbf{x}, \boldsymbol{\theta}) = \beta_0^l + \sum_{j=1}^{m} \beta_j^l B_j(\mathbf{x}, \mathbf{w}_j),\ l = 1, 2, ..., J$, where β_j^l is the weight of the connection between the hidden node j and the output node l. The transfer function of all hidden and output nodes is the identity function. We consider the softmax activation function given by:

$$g_l(\mathbf{x}, \boldsymbol{\theta}_l) = \frac{\exp f_l(\mathbf{x}, \boldsymbol{\theta}_l)}{\sum_{l=1}^{J} \exp f_l(\mathbf{x}, \boldsymbol{\theta}_l)},\ l = 1, 2, ..., J\ . \tag{1}$$

where $\boldsymbol{\theta}_l = (\boldsymbol{\beta}^l, \mathbf{w}_1, ..., \mathbf{w}_m)$, $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_J)$ and $\boldsymbol{\beta}^l = (\beta_0^l, \beta_1^l, ..., \beta_m^l)$. It interesting to note that the model can be regarded as the feed-forward computation of a three-layer neural network where the activation function of each hidden units is $\exp(t) = e^t$ and where we have to do a logarithmic transformation of the input variables x_i, [11].

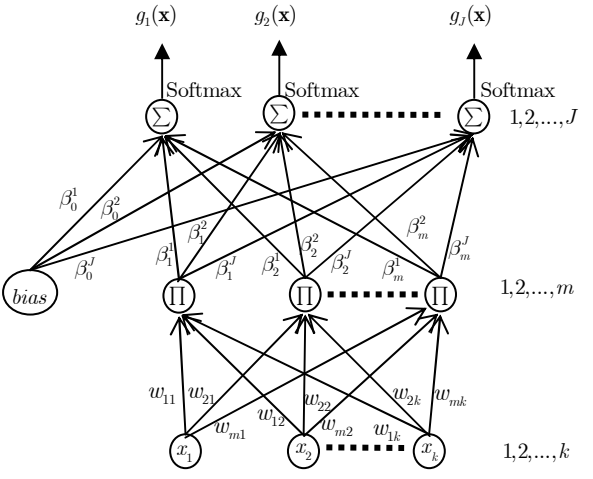

Fig. 1. Model of a product-unit based neural network

3 Classification Problem

In a classification problem, measurements x_i, $i = 1, 2, ..., k$, are taken on a single individual (or object), and the individuals have to be classified into one of the J classes based on these measurements. A training sample $D = \{(\mathbf{x}_n, \mathbf{y}_n); n = 1, 2, ..., N\}$ is available, where $\mathbf{x}_n = (x_{1n}, ..., x_{kn})$ is the random vector of measurements taking values in $\Omega \subset \mathbb{R}^k$, and $\mathbf{y}_n$ is the class level of the n-th individual. We adopt the common technique of representing the class levels using a "1-of-J" encoding vector $\mathbf{y} = \left(y^{(1)}, y^{(2)} ..., y^{(J)}\right)$, such as $y^{(l)} = 1$ if $\mathbf{x}$ corresponds to an example belonging to class l; otherwise, $y^{(l)} = 0$. Based on the training sample we try to find a decision function $C : \Omega \to \{1, 2, ..., J\}$ for classifying the individuals. A misclassification occurs when the decision rule C assigns an individual to a class j, when it actually comes from a class $l \neq j$. We define the corrected classified rate by $CCR = \frac{1}{N}\sum_{n=1}^{N} I(C(\mathbf{x}_n) = \mathbf{y}_n)$, where $I(\bullet)$ is the zero-one loss function. A good classifier tries to achieve the highest possible CCR in a given problem. We define the cross-entropy error function for the training observations as:

$$l(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{l=1}^{J} y_n^{(l)} \log g_l(\mathbf{x}_n, \boldsymbol{\theta}_l) = \frac{1}{N}\sum_{n=1}^{N}\left[-\sum_{l=1}^{J} y_n^{(l)} f_l(\mathbf{x}_n, \boldsymbol{\theta}_l) + \log \sum_{l=1}^{J} \exp f_l(\mathbf{x}_n, \boldsymbol{\theta}_l)\right] \quad (2)$$

The Hessian matrix of the error function $l(\boldsymbol{\theta})$ is, in general, indefinite and the error surface associated with the model is very convoluted with numerous local optimums. Moreover, the optimal number of basis functions of the model (i.e. the number of

hidden nodes in the neural network) is unknown. Thus, the estimation of the vector parameters $\hat{\boldsymbol{\theta}}$ is carried out by means an evolutionary algorithm (see Section 4). The optimum rule $C(\mathbf{x})$ is the following: $C(\mathbf{x}) = \hat{l}$, where $\hat{l} = \arg\max_l g_l(\mathbf{x}, \hat{\boldsymbol{\theta}})$, for $l = 1, ..., J$. Observe that softmax transformation produces positive estimates that sum to one and therefore the outputs can be interpreted as the conditional probability of class membership and the classification rule coincides with the optimal Bayes rule. On the other hand, the probability for one of the classes does not need to be estimated, because of the normalization condition. Usually, one activation function is set to zero and we reduce the number of parameters to estimate. Therefore, we set $f_J(\mathbf{x}, \boldsymbol{\theta}_J) = 0$.

4 Evolutionary Product-Unit Neural Networks

In this paragraph we carry out the evolutionary product-unit neural networks algorithm (EPUNN) to estimate the parameter that minimizes the cross-entropy error function. We build an evolutionary algorithm to design the structure and learn the weights of the networks. The search begins with an initial population of product-unit neural networks, and, in each iteration, the population is updated using a population-update algorithm. The population is subjected to the operations of replication and mutation. Crossover is not used due to its potential disadvantages in evolving artificial networks. With these features the algorithm falls into the class of evolutionary programming .The general structure of the EA is the following:

(1) Generate a random population of size N_p.

(2) Repeat until the stopping criterion is fulfilled
   (a) Calculate the fitness of every individual in the population.
   (b) Rank the individuals with respect to their fitness.
   (c) The best individual is copied into the new population.
   (d) The best 10% of population individuals are replicated and substitute the worst 10% of individuals.

   Over that intermediate population we:
   (e) Apply parametric mutation to the best 10% of individuals.
   (f) Apply structural mutation to the remaining 90% of individuals.

We consider $l(\boldsymbol{\theta})$ being the error function of an individual g of the population. Observe that g can be seen as the multivaluated function $g(\mathbf{x}, \boldsymbol{\theta}) = (g_1(\mathbf{x}, \boldsymbol{\theta}_1), ..., g_l(\mathbf{x}, \boldsymbol{\theta}_l))$. The fitness measure is a strictly decreasing transformation of the error function $l(\boldsymbol{\theta})$ given by $A(g) = (1 + l(\boldsymbol{\theta}))^{-1}$. Parametric mutation is accomplished for each coefficient w_{ji}, β_j^l of the model with Gaussian noise: $w_{ji}(t+1) = w_{ji}(t) + \xi_1(t)$, $\beta_j^l(t+1) = \beta_j^l(t) + \xi_2(t)$, where $\xi_k(t) \in N(0, \alpha_k(t))$, $k = 1, 2$, represents a one-dimensional normally-distributed random variable with mean 0 and variance $\alpha_k(t)$, where $\alpha_1(t) < \alpha_2(t)$, and t is the t-th generation. Once the mutation is performed, the fitness of the individual is recalculated and the usual

simulated annealing is applied. Thus, if ΔA is the difference in the fitness function after and preceding the random step, the criterion is: if $\Delta A \geq 0$ the step is accepted, if $\Delta A < 0$, the step is accepted with a probability $\exp(\Delta A / T(g))$, where the temperature $T(g)$ of an individual g is given by $T(g) = 1 - A(g),\ 0 \leq T(g) < 1$.

The variance $\alpha_k(t)$ is updated throughout the evolution of the algorithm. There are different methods to update the variance. We use the 1/5 success rule of Rechenberg, one of the simplest methods. This rule states that the ratio of successful mutations should be 1/5. Therefore, if the ratio of successful mutations is larger than 1/5, the mutation deviation should increase; otherwise, the deviation should decrease. Thus:

$$\alpha_k(t+s) = \begin{cases} (1+\lambda)\alpha_k(t) & \text{if } s_g > 1/5 \\ (1-\lambda)\alpha_k(t), & \text{if } s_g < 1/5 \\ \alpha_k(t) & \text{if } s_g = 1/5 \end{cases} \tag{3}$$

where $k = 1,2$, s_g is the frequency of successful mutations over s generations and $\lambda = 0.1$. The adaptation tries to avoid being trapped in local minima and also to speed up the evolutionary process when searching conditions are suitable.

Structural mutation implies a modification in the neural network structure and allows explorations of different regions in the search space while helping to keep up the diversity of the population. There are five different structural mutations: node deletion, connection deletion, node addition, connection addition and node fusion. These five mutations are applied sequentially to each network. In the node fusion, two randomly selected hidden nodes, a and b, are replaced by a new node, c, which is a combination of both. The connections that are common to both nodes are kept, with weights given by: $\beta_c^l = \beta_a^l + \beta_b^l, w_{jc} = \frac{1}{2}(w_{ja} + w_{jb})$. The connections that are not shared by the nodes are inherited by c with a probability of 0.5 and its weight is unchanged. In our algorithm, node or connection deletion and node fusion is always attempted before addition. If a deletion or fusion mutation is successful, no other mutation will be made. If the probability does not select any mutation, one of the mutations is chosen at random and applied to the network.

5 Experiments

The parameters used in the evolutionary algorithm are common for the four problems. We have considered $\alpha_1(0) = 0.5$, $\alpha_2(0) = 1$, $\lambda = 0.1$ and $s = 5$. The exponents w_{ji} are initialized in the $[-5,5]$ interval, the coefficients β_j^l are initialized in $[-5,5]$. The maximum number of hidden nodes is $m = 6$. The size of the population is $N_P = 1000$. The number of nodes that can be added or removed in a structural mutation is within the $[1,2]$ interval. The number of connections that can be added or removed in a structural mutation is within the $[1,6]$ interval. The stop criterion is reached if the following condition is fulfilled: for 20 generations there is no

improvement either in the average performance of the best 20% of the population or in the fitness of the best individual. We have done a simple linear rescaling of the input variables in the interval $[1,2]$, being X_i^* the transformed variables.

We evaluate the performance of our method on four data sets taken from the UCI repository [7]. For every dataset we performed ten runs of ten-fold stratified cross-validation. This gives a hundred data points for each dataset, from which the average classification accuracy and standard deviation is calculated. Table 1 shows the statistical results over 10 runs for each fold of the evolutionary algorithm for the four data sets. With the objective of presenting an empirical evaluation of the performance of the EPUNN method, we compare our approach to the most recent results [12] obtained using different methodologies (see Table 2). Logistic model tree, LMT, to logistic regression (with attribute selection, SLogistic, and for a full logistic model, MLogistic); induction trees (C4.5 and CART); two logistic tree algorithms: LTreeLog and finally, multiple-tree models M5´ for classification, and boosted C4.5 trees using AdaBoost.M1 with 10 and 100 boosting interactions. We can see that the results obtained by EPUNN, with architectures (13:2:2), (34:3:2), (4:5:3) and (51:3:2) for each data set, are competitive with the learning schemes mentioned previously.

Table 1. Statistical results of training and testing for 30 executions of EPUNN model

Data set	CCR_T				CCR_G				#connect		#node
	Mean	SD	Best	Worst	Mean	SD	Best	Worst	Mean	SD	
Heart-stat	84.65	1.63	88.48	80.25	**81.89**	6.90	**96.30**	62.96	14.78	3.83	2
Ionosphere	93.79	1.46	97.15	90.19	**89.63**	5.52	**100**	74.29	43.97	13.87	3
Balance	97.26	0.98	99.47	94.32	**95.69**	2.36	**100**	90.32	25.62	2.18	5
Australian	87.01	0.82	88.57	85.02	**85.74**	3.90	**95.65**	78.26	44.13	16.26	3

Table 2. Mean classification accuracy and standard deviation for: LMT, SLogistic, MLogistic, C4.5, CART, NBTree, LTreeLin, LTreeLog, M5', ABOOST and EPUNN method

Data set	LMT	SLogistic	MLogistic	C4.5	CART	NBTree
Heart-stat	83.22±6.50	83.30±6.48	**83.67±6.43**	78.15±7.42	78.00±8.25	80.59±7.12
Ionosphere	92.99±4.13	87.78±4.99	87.72±5.57	89.74±4.38	89.80±4.78	89.49±5.12
Balance	89.71±2.68	88.74±2.91	89.44±3.29	77.82±3.42	78.09±3.97	75.83±5.32
Australian	85.04±3.84	85.04±3.97	85.33±3.85	85.57±3.96	84.55±4.20	85.07±4.03

Data set	LTreeLin	LTreeLog	M5'	ABoost(10)	ABoost(100)	EPUNN	W/L
Heart-stat	83.52±6.28	83.00±6.83	82.15±6.77	78.59±7.15	80.44±7.08	81.89±6.90	**5/6**
Ionosphere	88.95±5.10	88.18±5.06	89.92±4.18	93.05±3.92	**94.02±3.83**	89.63±5.52	**7/4**
Balance	**92.86±3.22**	92.78±3.49	87.76±2.23	78.35±3.78	76.11±4.09	**95.69±2.36**	**11/0**
Australian	84.99±3.91	84.64±4.09	85.39±3.87	84.01±4.36	**86.43±3.98**	85.74±3.90	**10/1**

6 Conclusions

We propose a classification method that combines a nonlinear model, based on a special class of feed-forward neural network, namely product-unit neural networks,

and an evolutionary algorithm that finds the optimal structure of the model and estimates the corresponding parameters. Up to now, the studies on product units have been applied mainly to solve regression problems and have not addressed the problem of the design of both structure and weights simultaneously in this kind of neural network, either using classic or evolutionary based methods. Our approach uses softmax transformation and the cross-entropy error function. From a statistical point of view, the approach can be seen as nonlinear multinomial logistic regression, where optimization of the log-likelihood is made by using evolutionary computation. The empirical results show that the evolutionary product-unit model performs well compared to other learning classification techniques. We obtain very promising results in terms of classification accuracy and the complexity of the classifier.

References

[1] T. J. Hastie and R. J. Tibshirani, Generalized Additive Models. London: Chapman & Hall, 1990.

[2] C. Kooperberg, S. Bose, and C. J. Stone, "Polychotomous Regression," Journal of the American Statistical Association, vol. 92, pp. 117-127, 1997.

[3] R. Durbin and D. Rumelhart, "Products Units: A computationally powerful and biologically plausible extension to backpropagation networks," Neural Computation, vol. 1, pp. 133-142, 1989.

[4] A. C. Martínez-Estudillo, F. J. Martínez-Estudillo, C. Hervás-Martínez, et al., "Evolutionary Product Unit based Neural Networks for Regression," Neural Networks, pp. 477-486, 2006.

[5] A. C. Martínez-Estudillo, C. Hervás-Martínez, A. C. Martínez-Estudillo, et al., "Hybridation of evolutionary algorithms and local search by means of a clustering method," IEEE Transactions on Systems, Man and Cybernetics, Part. B: Cybernetics, vol. 36, pp. 534-546, 2006.

[6] X. Yao, "Evolving artificial neural network," Proceedings of the IEEE, vol. 9 (87), pp. 1423-1447, 1999.

[7] C. Blake and C. J. Merz, " UCI repository of machine learning data bases," www.ics.uci.edu/ mlearn/MLRepository.thml, 1998.

[8] M. Schmitt, "On the Complexity of Computing and Learning with Multiplicative Neural Networks," Neural Computation, vol. 14, pp. 241-301, 2001.

[9] A. Ismail and A. P. Engelbrecht, "Global optimization algorithms for training product units neural networks," presented at International Joint Conference on Neural Networks IJCNN`2000, Como, Italy, 2000.

[10] A. P. Engelbrecht and A. Ismail, "Training product unit neural networks," Stability and Control: Theory and Applications, vol. 2, pp. 59-74, 1999.

[11] K. Saito and R. Nakano, "Extracting Regression Rules From Neural Networks," Neural Networks, vol. 15, pp. 1279-1288, 2002.

[12] N. Landwehr, M. Hall, and F. Eibe, "Logistic Model Trees," Machine Learning, vol. 59, pp. 161-205, 2005.

Answers to the Referees

Referee 1

The functional model defined by the product-unit neural network is given by

$$(1)\quad f_l(\mathbf{x},\boldsymbol{\theta}) = \beta_0^l + \sum_{j=1}^{m} \beta_j^l B_j(\mathbf{x},\mathbf{w}_j),\ \ l = 1,2,...,J$$

where $B_j(\mathbf{x},\mathbf{w}_j) = \prod_{i=1}^{k} x_i^{w_{ji}}$. We are agreeing with the referee 1 that this expression is analytically equivalent to the expression:

$$(2)\quad f_l(\mathbf{x},\boldsymbol{\theta}) = \beta_0^l + \sum_{j=1}^{m} \beta_j^l \exp\left(\sum_{j=1}^{n} w_{ji} \ln(x_i)\right),\ \ l = 1,2,...,J$$

This equation can be regarded as the feed-forward computation of a three-layer neural network where the activation function of each hidden units is $\exp(t) = e^t$ and where we have done a logarithmic transformation inputs variables x_i . Therefore we can have different architectures of the same functional model. We have chosen the architecture associated to the first equation because it is easier.

However, the functional model given by (1), or equivalent by (2), is different from the functional model of MLP. The function that a classic multilayer feed-forward network computes is

$$(3)\quad f_l(\mathbf{x},\boldsymbol{\theta}) = \beta_0^l + \sum_{j=1}^{m} \beta_j^l \sigma\left(\sum_{j=1}^{n} w_{ji} x_i - \theta_j\right),\ \ l = 1,2,...,J$$

where σ is a sigmoidal function.

From an analytical point of view, the model defined by (3) is different to the (1) and (2) ones.

We have included in the page 3 of the paper a brief comment about this question.

Referee 2

There are a reduced number of papers dealing with product-unit neural networks. For this, the references could seem a bit outdated. We have included in the paper more recent references ([4], [5], [11]) to solve this question. On the other hand, a deeper analysis about the "why" of the motivation on the problem and the theoretical justification of the product-unit approach would need a long and extensive paper to explain the VC dimension of the PU and the upper bounds of the VC dimension obtained in Schmitt [10].