A Feedforward Constructive Neural Network Algorithm for Multiclass Tasks Based on Linear Separability

João Roberto Bertini Jr. and Maria do Carmo Nicoletti

Abstract. Constructive neural network (CoNN) algorithms enable the architecture of a neural network to be constructed along with the learning process. This chapter describes a new feedforward CoNN algorithm suitable for multiclass domains named MBabCoNN, which can be considered an extension of its counterpart BabCoNN, suitable for two-class classification tasks. Besides describing the main concepts involved in the MBabCoNN proposal, the chapter also presents a comparative analysis of its performance *versus* the multiclass versions of five well-known constructive algorithms, in eight knowledge domains, as empirical evidence of the MBabCoNN suitability and efficiency for multiclass classification tasks.

Keywords: Constructive neural network algorithm, LS-discriminant learning, Barycentric Correction Procedure, Multiclass classification.

1 Introduction

There are many different methods that allow the automatic learning of concepts, as can be seen in [1] and [2]. One particular class of relevant machine learning methods is based on the concept of linear separability (LS).

The concept of linear separability permeates many areas of knowledge and based on the definition given in [3] it can be stated as: Let E be a finite set of N distinct patterns $\{E_1, E_2, \ldots, E_N\}$, each pattern E_i $(1 \leq i \leq N)$ described as $E_i = \langle x_1,\ldots,x_k\rangle$, where k is the number of attributes that defines a pattern. Let the patterns of E be classified in such a way that each pattern in E belongs to only one of the M classes C_j $(1 \leq j \leq M)$. This classification divides the set of patterns E into

João Roberto Bertini Jr.
Institute of Mathematics and Computer Science, University of São Paulo,
Av. Trabalhador São Carlense 400, São Carlos, Brazil
bertini@icmc.usp.br

Maria do Carmo Nicoletti
Department of Computer Science, Federal University of São Carlos,
Via Washington Luiz, km. 238, São Carlos, Brazil
carmo@dc.ufscar.br

L. Franco et al. (Eds.): Constructive Neural Networks, SCI 258, pp. 145–169.
springerlink.com

the subsets EC_1, EC_2, …, EC_M, such that each pattern in EC_i belongs to class C_i, for i = 1, …, *M*. If a linear machine can classify the patterns in *E* into the proper class, the classification of *E* is a linear classification and the subsets EC_1, EC_2, …, EC_M are linearly separable. Stated another way, a classification of *E* is linear and the subsets EC_1, EC_2, …, EC_M, are linearly separable if and only if linear discriminant functions g_1, g_2, …, g_M exist such that

$$g_i(E) > g_j(E) \qquad \text{for all } E \in EC_i$$
$$j = 1, \ldots, M, j \neq i \qquad \text{for all } i = 1, \ldots, M$$

Since the decision regions of a linear machine are convex, if the subsets EC_1, EC_2, …, EC_M are linearly separable, then each pair of subsets EC_i, EC_j, i, j = 1, …, *M*, i ≠ j, is also linearly separable. That is, if EC_1, EC_2, …, EC_M, are linearly separable, then EC_1, EC_2, …, EC_M, are also pairwise linearly separable.

According to Elizondo [4], linearly separable based learning methods can be divided into four groups. Depending on their main focus they may be based on linear programming, computational geometry, neural networks or quadratic programming.

This chapter describes a new neural network algorithm named MBabCoNN (Multiclass Barycentric-based Constructive Neural Network) suitable for multiclass classification problems. The algorithm incrementally constructs a neural network by adding hidden nodes that linearly separate sub-regions of the feature space. It can be considered a multiclass version of the two-class CoNN named BabCoNN (Barycentric-based Constructive Neural Network) proposed in [5].

The chapter is an extended version of an earlier paper [6] and is organized as follows. Section 2 stresses the importance of CoNN algorithms and discusses the role played by the algorithm used for training individual Threshold Logic Units (TLU), particularly focusing on the BCP algorithm [7]. Section 3 highlights the main characteristics of the five well-known CoNN multiclass algorithms used in the empirical experiments described in Section 5. Section 4 initially outlines the basic features of the two-class BabCoNN algorithm briefly presenting the main concepts and strategies used by BabCoNN when learning and classifying and,presents a detailed description of the multiclass MBabCoNN algorithm divided into two parts: learning the neural network and using the network learnt for classifying previously unseen patterns. Section 5 presents and discusses the results of 16 algorithms; four of them are versions of PRM and BCP for multiclass tasks and the other 12 are variants of the basic multiclass algorithms used, namely: MTower, MPyramid, MUpstart, MTiling, MPerceptron-Cascade and MBabCoNN in eight knowledge domains from the UCI Repository [8]. The Conclusion section ends the chapter by presenting a summary of the main results highlighting a few possible research lines to investigate, aiming at improving the MBabCoNN algorithm.

2 Constructive NN and the Relevance of TLU Training

Whereas conventional neural network (NN) training algorithms such as the Backpropagation algorithm require the NN architecture to be defined before learning

can begin, constructive neural network (CoNN) algorithms allow the network architecture to be constructed simultaneously with the learning process; both subprocesses, learning and constructing the network, are interdependent.

Constructive neural network (CoNN) algorithms do not assume fixed network architecture before training begins. The main characteristic of a CoNN algorithm is the dynamic construction of the network's hidden layer(s), which occurs simultaneously with training. A description of a few well-known CoNN algorithms can be found in [9] and [10]; among the most well-known are: Tower and Pyramid [11], Tiling [12], Upstart [13], Perceptron-Cascade [14], Pti and Shift [15].

2.1 Training Individual TLUs

Usually the basic function performed by a CoNN algorithm is the addition to the network architecture of a new TLU and its subsequent training. For this reason CoNN algorithms are very dependent on the TLU training algorithm used. For training a TLU, a constructive algorithm generally employs the Perceptron or any of its variants, such as Pocket or Pocket with Ratchet Modification (PRM) [11]. Considering that CoNN algorithms depend heavily on an efficient TLU training algorithm, there is still a need for finding new and better methods, although some of the Perceptron variants (especially the PRM) have been widely used with good results.

The Barycentric-based Correction Procedure (BCP) algorithm [7] [16], although not widely adopted, has performed well when used for training individual TLUs (see [17] for instance) and has established itself as a good competitor compared to the PRM when used by CoNN algorithms (see [18] for a performance comparison). Good results have also been obtained by allowing both algorithms (BCP and PRM) to compete for training the next neuron to be added to the network; the proposal of this hybrid constructive algorithm and its results can be found in [19]. In spite of BCP being poorly explored in the literature, its good performance motivated the choice of this algorithm as the TLU's training algorithm embedded in both the BabCoNN and its multiclass version MBabCoNN, described in this chapter.

The BCP is based on the geometric concept of the barycenter of a convex hull and the algorithm (for a two-class problem) iteratively calculates the barycenters of the regions defined by the positive and the negative training patterns. Unlike Perceptron based algorithms, this algorithm calculates the weight vector and the bias separately. The BCP defines the weight vector as the vector that connects two points: the barycenter of the convex hull of positive patterns (class +1) and the barycenter of the convex hull defined by negative patterns (class −1). The convex hull of an n-dimensional set of points E is the intersection of all convex sets containing E, and the barycenter stands for its center of mass [20]. It follows a brief overview of the BCP algorithm.

Let $E = E1 \cup E2$ be a training set such that E1 is the subset of training patterns with class 1 and E2 the set of training patterns with class −1, and let $|E1| = k_1$ and $|E2| = k_2$. The barycenters b_1 and b_2 represent the center of mass of the convex hull formed by patterns belonging to each class, respectively. In the algorithm they are defined as the weighted averages of patterns in E1 and E2 respectively as described by eq. (1),

$$b_1 = \frac{\sum_{i=1}^{k_1} \alpha_i E1^i}{\sum_{i=1}^{k_1} \alpha_i} \quad \text{and} \quad b_2 = \frac{\sum_{i=1}^{k_2} \mu_i E2^i}{\sum_{i=1}^{k_2} \mu_i} \tag{1}$$

where α and μ are weight vectors, $\alpha = \langle \alpha_1, \alpha_2, \ldots, \alpha_{k_1} \rangle$ and $\mu = \langle \mu_1, \mu_2, \ldots, \mu_{k_2} \rangle$, responsible for modifying the position of the barycenters. For the experiments described in Section 5, both weight vectors were randomly initialized in the range [1,2], as recommended in [17]. They are used to vary the barycenters, at each execution, increasing the probability of finding a better weight vector.

In the BCP procedure the weight vector is defined as $W = b_1 - b_2$ and the hyperplane it defines is given by $W.x + \theta = 0$, where θ is the bias term. Once W is determined, the bias term θ is separately defined according to the following procedure. Let p be a pattern and consider the function $V: R^n \rightarrow R$ given by eq. (2).

$$V(p) = -W.p \tag{2}$$

Consider subsets $V1 = \{V(p) \mid p \in E1\}$ and $V2 = \{V(p) \mid p \in E2\}$ and let $V = V1 \cup V2$. The greatest and the smallest values of V1 and V2 are then determined. If $\max(V1) < \min(V2)$, the training set is linearly separable and θ is chosen such that $\max(V1) < \theta < \min(V2)$ and the algorithm ends. Otherwise either the set is not linearly separable or the current weight vector is not correctly positioned.

If $\max(V1) \geq \min(V2)$ the chosen value for θ should minimize the misclassifications. To do so, consider the set $Ex = \{ext1, ext2, ext3, ext4\}$ whose values correspond to the smallest and biggest values of V1 and V2 respectively. Consider $P_- = [ext1,ext2) \cap V$; $P_+ = [ext3,ext4) \cap V$ and $P_{ov} = [ext2,ext3) \cap V$. As sets P_- and P_+ have patterns belonging to only one class, they are called exclusion zones. Since P_{ov} has patterns belonging to both classes it is called the overlapping zone. To choose an appropriate bias, the algorithm iteratively establishes its value as the arithmetic mean of two consecutive values in the overlapping zone. The value that correctly classifies the greatest number of patterns is chosen as bias.

At a certain, iteration let R and S be the sets of patterns belonging to classes 1 and −1 respectively and let b_1 and b_2 be the barycenters of region R and S respectively (calculated as in eq. (1)). Let $RE \subset R$ and $SE \subset S$ be the subsets of misclassified patterns. The algorithm determines the barycenters be_1' and be_2' of RE and SE respectively and then, creates two vectors $e_1 = b_1 - be_1$' and $e_2 = b_2 - be_2$'. The two vectors are then multiplied by random values from [0,1], say r_1 and r_2, giving rise to the new barycenters $b_1' = r_1.e_1$ and $b_2' = r_2.e_2$. A new weight vector W' (and consequently the new hyperplane H'), is then obtained by connecting the new barycenters. The process continues while wrongly classified patterns remain or the number of iterations has not reached its predefined value. Due to its geometric approach, the BCP ends after a few iterations and the final hyperplane tends to be a good separator between the two classes, even in situations where the training set is not linearly separable.

3 Reviewing Five Well-Known Multiclass CoNN Algorithms

Multiclass classification tasks are common in pattern recognition. Frequently a classification task with M (> 2) classes is treated as M two-class tasks. Although this approach may be suitable for some applications, there is still a need for more effective ways of dealing with multiclass problems. CoNNs have proved to be a good alternative for two-class tasks and have the potential to become good alternatives for multiclass domains as well.

Multiclass constructive algorithms start by training as many output neurons as there are classes in the training set; generally two different strategies can be employed for the task, the *independent* (I) and the *winner-takes-all* (WTA). As stated in [21] in the former strategy each output neuron is trained independently of the others. The WTA strategy, however, explores the fact that the membership of a pattern in one class prevents its belonging to any other class. Using the WTA strategy, for any pattern, the output neuron with the highest net input is assigned an output of 1 and all other neurons are assigned outputs of −1. In the case of a tie for the highest net input all neurons are assigned an output of −1, thereby rendering the pattern incorrectly classified.

The main goal of this section is to provide a brief overview of the five well-known multiclass CoNN algorithms used in the experiments (for a more detailed description see [9]) described in Section 5 and to describe the new multiclass algorithm MBabCoNN. So far, multiclass problems have not been the main focus of CoNN research and consequently most of the multiclass algorithms available are extensions of their two-class counterparts.

The multiclass MTower algorithm was proposed in [22] and can be considered a direct extension of the two-class Tower algorithm. The Tower creates a NN with only one TLU per hidden layer. In a Tower network [11] each new hidden neuron introduced is connected to all the input neurons and to the hidden neuron previously created – this causes the network to resemble a tower. Similarly to the two-class Tower, the MTower adds TLUs to the network; instead of one at a time, like the Tower, it adds as many hidden neurons as there are classes.

For an M-class problem, the MTower adds M hidden neurons per hidden layer. Each one of the M neurons in a certain hidden layer has connections with all the neurons in the input layer as well as connections with all the M neurons of the previously added hidden layer. The addition of new layers to the network ends when any of the following stopping criteria is satisfied: (1) the current network correctly classifies all the training patterns; (2) the threshold on the number of layers has been reached; (3) the current network accuracy is worse than the accuracy of the previous network (i.e., the current network without the addition of the last hidden layer). If (3) happens the algorithm removes the last layer added and ends the process, returning the network constructed so far.

The multiclass MPyramid, also proposed in [22], is a direct extension of its two-class counterpart Pyramid algorithm, described in [11]. MPyramid extends the Pyramid simply by adding M hidden neurons per layer (corresponding to the existing M classes in the training set) instead of only one at each step. The difference between the Tower and Pyramid algorithms (and consequently between their

M-class versions) lies on the connections. In a Pyramid network each newly added hidden neuron has connections with all the previously added hidden ones as well as with the input neurons.

The two-class Upstart algorithm [13] constructs the neural network as a binary tree of TLUs and it is governed by the addition of new hidden neurons, specialized in correcting *wrongly-on* or *wrongly-off* errors made by the previously added neurons. A natural extension of this algorithm for multiclass tasks would be an algorithm that constructs *M* binary trees, each one responsible for the learning of one of the *M* classes found in the training set. This approach, however, would not take into account a possible relationship that might exist between the *M* different classes. The MUpstart proposal, described in [23], tries to circumvent the problem by grouping the created hidden neurons in a single hidden layer. Each hidden neuron is created aiming at correcting the most frequent error (*wrongly-on* or *wrongly-off*) committed by a single neuron among the *M* output neurons. The hidden neurons are trained with patterns labeled with two classes only and they can fire 0 or 1. Each hidden neuron is directly connected to every neuron in the output layer. The input layer is connected to the hidden neurons as well as to the output neurons.

The Tiling algorithm [12] constructs a neural network where hidden nodes are added to a layer in a similar fashion to laying tiles. Each hidden layer in a Tiling network has a master neuron and a few ancillary neurons. The output layer has only one master neuron. Tiling constructs a neural network in successive layers such that each new layer has a smaller number of neurons than the previous layer. Similarly to this approach, the MTiling method, as proposed in [24], constructs a multi-layer neural network where the first hidden layer has connections to the input layer and each subsequent hidden layer has connections only to the previous hidden layer. Each layer has master and ancillary neurons with the same functions they perform in a Tiling network i.e., the master neurons are responsible for classifying the training patterns and the ancillary ones are responsible for making the layer faithful. The role played by the ancillary neurons in a hidden layer is to guarantee that the layer does not produce the same output for any two training patterns belonging to different classes. In the MTiling version the process of adding a new layer is very similar to the one implemented by Tiling. However, while the Tiling algorithm adds only one master neuron per layer, the MTiling adds *M* master neurons (where *M* is the number of different classes in the training set).

The Perceptron Cascade algorithm [14] is a neural constructive algorithm that constructs a neural network with an architecture resembling the one constructed by the Cascade Correlation algorithm [25] and it uses the same approach for correcting the errors adopted by the Upstart algorithm [13]. Unlike the Cascade Correlation however, the Perceptron Cascade uses the Perceptron (or any of its variants) for training individual TLUs). Like the Upstart algorithm, the Perceptron Cascade starts the construction of the network by training the output neuron and hidden neurons are added to the network similarly to the process adopted by the Cascade Correlation: each new neuron is connected to both the output and input neurons and has connections with all hidden neurons previously added to the network. The MPerceptron-Cascade version, proposed in [22], is very similar to the

MUpstart described earlier in this section, the main difference between them being that the neural network architecture induced by both. The MPerceptron-Cascade adds the new hidden neurons in new layers while the MUpstart adds them in one single layer.

4 The Multiclass MBabCoNN Proposal

The MBabCoNN proposal can be considered an extension of the two-class algorithm called BabCoNN [5], suitable for classification tasks involving $M > 2$ classes. In order to present and discuss the MBabCoNN proposal, this section initially presents a brief description of the most important features of the BabCoNN algorithm, paying particular attention to the mechanism employed by the hidden neurons for firing their outputs, since the MBabCoNN shares the same strategy. To facilitate the understanding of MBabCoNN, the learning and the classification processes implemented by the algorithm are approached separately; the trace of both processes is shown via an example.

4.1 The Two-Class BabCoNN Algorithm

BabCoNN is a new proposal that borrows some ideas from the BCP to build a neural network. Like Upstart, Perceptron Cascade (PC) and Shift, BabCoNN also constructs the network beginning with the output neuron. However, it creates only one hidden layer; each hidden neuron is connected to the input layer as well as to output neuron, like the Shift algorithm [15]. Unlike Shift however, the connections created by BabCoNN do not have an associated weight. The Upstart, PC and Shift algorithms construct the network by adding new hidden neurons specialized in correcting *wrongly-on* or *wrongly-off* errors. The BabCoNN, however, employs a different strategy to add new hidden neurons to the network.

Network construction starts by training the output neuron, using the BCP. Next, the algorithm identifies all the misclassified training patterns; if there are none, the algorithm stops, otherwise it starts adding neurons (one at a time) to the single hidden layer of the network, in order not to have misclassified patterns. A hidden neuron added to the hidden layer will be trained with the training patterns that were misclassified by the last added neuron; the first hidden neuron will be trained with the patterns that were misclassified by the output neuron; the second hidden neuron will be trained with the set containing the patterns that the first hidden neuron was unable to classify correctly, and so on. The process continues up to the point where no training patterns remain or all the remaining patterns belong to the same class.

The process of building the network architecture is described by the pseudocode given in Fig. 1, where $E = \{E_1, E_2,\ldots E_N\}$ represents the training set and each training pattern is described as $E_i = \langle x_1, x_2,\ldots x_k, C\rangle$, i.e., k attributes and an associated class $C \in \{-1, 1\}$.

In Fig. 1 the variables *output* and *hiddenLayer[ ]* define the neural network. The variable *output* represents a single neuron, and *hiddenLayer[ ]* is a vector

representing the hidden neurons. The function *bcp( )* stands for the BCP algorithm, used for training individual neurons. The function *removeClassifiedPatterns( )* removes from the training set the patterns that were correctly classified by the last added neuron and *bothClasses( )* is a Boolean function that returns 'true' if the current training set still has patterns belonging to both classes and 'false' otherwise.

Due to the way the learning phase is conducted by BabCoNN, each hidden neuron of the network is trained using patterns belonging to a region of the training space (i.e., the one defined by the patterns that were misclassified by the previous hidden neuron added to the network). This particular aspect of the algorithm has the effect of introducing an undesirable 'redundancy', in the sense that a pattern may be correctly classified by more than one hidden neuron. This has been sorted out by implementing a classification process where the neurons of the hidden layer have a particular way of firing their output.

```
procedure BabCoNN_learner(E)
begin
   output ← bcp(E)
   nE ← removeClassifiedPatterns(E)
   h ← 0
   while bothClasses(nE) do
    begin
      h ← h + 1
      hiddenLayer[h] ← bcp(nE)
      nE ← removeClassifiedPatterns(nE)
    end
end procedure.
```

Fig. 1 BabCoNN algorithm for constructing a neural network.

Given an input pattern to be classified by a BabCoNN network, each hidden neuron has three possible outputs: 1, when the input pattern is classified as positive; −1, when the pattern is classified as negative and 0, when the pattern is classified as undetermined. Aiming at stressing the classification power of the hidden neurons, as well as providing a way for them to deal with unknown patterns, a limited 'region of action' is assigned to each hidden neuron. The region is limited by two thresholds associated to each hidden neuron, one for the positive class and the other for the negative class. The threshold values are determined as the largest Euclidean distance between the barycenter of a given class and the patterns of the same class are used to train the current neuron.

Figure 2 illustrates the process. The two-dimensional patterns used for training the hidden neuron are represented by '+' (positive) and '–' (negative); b_1 and b_2 are the barycenters of the regions defined by the '+' and the '–' patterns respectively; W is the weight vector after the training and H is the hyperplane defined by

both, W and the bias. For each class, the region is defined as the hypersphere whose radius is given by the largest distance between all correctly classified patterns and the corresponding barycenter of the region.

To exemplify how a hidden neuron behaves during the classification phase, let each $Y_i = \langle y_{i_1}, y_{i_2} \rangle$, $i = \{1, 2, 3, 4\}$ be a given pattern to be classified. As can be seen in Figure 2, four situations may occur:

(1) The new pattern (Y_1) is in the positive classification region of the hidden neuron. The pattern Y_1 is classified as positive by the neuron, which fires +1;
(2) The new pattern (Y_2) is in the positive region, but now lying on the other side of the hyperplane; this would make the neuron classify Y_2 as negative. However, the neuron will fire the value 0 since there is no guarantee that the pattern is negative;
(3) The new pattern (Y_3) is not part of any region; in this case the neuron fires the value 0 independently of the classification given by the hyperplane it represents;
(4) The new pattern (Y_4) is in the negative classification region of the hidden neuron. The pattern Y_4 is classified as negative and the neuron fires the value −1. Note that the regions may overlap with each other and, eventually, a pattern may lie in both regions. When that happens, the hidden neuron (as implemented by the version used in the experiments described in Section 5) assigns the pattern the class is given by the hyperplane.

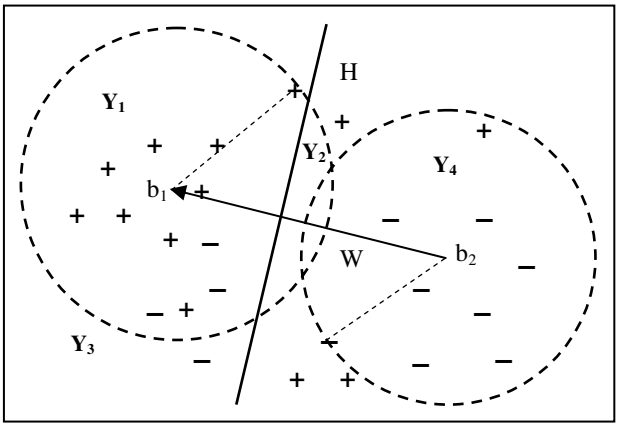


Fig. 2 BabCoNN hidden neuron firing process.

The pseudocode of the classification procedure is described in Fig. 3. After each hidden neuron fires its output, the output neuron decides which class the given pattern belongs to. The decision process is based on the sum of all the responses; if the resulting value is positive, the pattern is classified as positive, otherwise, as negative. If the sum result is 0, the output node is in charge of classifying the pattern.

The function *classification( )* returns the neuron classification (1 or –1), this is the usual classification that uses the weight vector and bias. Both functions *belongsToPositive( )* and *belongsToNegative( )* are Boolean functions. The first returns 'true' if the pattern lies in the positive region and 'false' otherwise. The second returns 'true' if the pattern lies in the negative region and 'false' otherwise. The *Hlc[ ]* vector stores the classifications given by all hidden neurons, for a given pattern and the last conditional command in the classification algorithm defines the class associated with the input pattern X.

```
procedure BabCoNN_classifier(X)
{X is the pattern to be classified}
begin
   for i ← 1 to h do
    begin
      C ← classification(hiddenLayer[i], X)
      Bp ← belongsToPositive(hiddenLayer[i], X)
      Bn ← belongsToNegative(hiddenLayer[i], X)
      if (C = 1 and Bp) then Hlc[i] ← 1
                        else if (C = –1 and Bn)
                                then Hlc[i] ← –1
                                else Hlc[i] ← 0
    end
   sum ← 0
   for j ← 1 to h do
      sum ← Hlc[j] + sum
   if sum ≠ 0 then sum / |sum|
              else classification(output,X)
end procedure.
```

Fig. 3 BabCoNN classification process.

4.2 The MBabCoNN Learning Algorithm

Figure 4 presents the pseudocode of the algorithm that implements the MBabCoNN learning process; the input to the algorithm is the training set E with patterns belonging to $M > 2$ classes. MBabCoNN can deal with Boolean, integer and real-valued tasks.

MBabCoNN constructs the network beginning with the output layer containing as many neurons as there are classes in the training set (each output neuron is associated to a class). The algorithm is flexible enough to allow the output neurons to be trained using any TLU algorithm combined with either strategy, independent or WTA.

After adding and training the M output neurons using procedure *MTluTraining( )*, the algorithm identifies the misclassifications the current network makes on the training set, via procedure *evaluateNetwork( )*, and starts to add neurons to its single hidden layer in order to correct the classification mistakes made by the output neurons.

In a MBabCoNN neural network each hidden neuron can be considered a two-class BabCoNN-like hidden neuron, i.e. it only fires 1, −1 or 0 values. In order to add a hidden neuron, MBabCoNN first finds which output neuron (class) is responsible for the greatest number of misclassifications in relation to patterns belonging to all the other output classes, via *highest_wrongly-on_error( )*, detailed in Figure 5.

A hidden neuron is then added to the hidden layer and is trained with a set containing patterns of two classes only: those belonging to the class the output neuron represents (which are relabeled as class −1) and those belonging to the misclassified class (which are relabeled as class 1).

Each newly added hidden neuron is then connected only to the two output neurons whose classes it separates. The connection to the neuron responsible for the misclassifications has weight 1 and the other −1. In fact the classes' labels can be arbitrarily chosen, the only proviso is that the weight must correspond to the relabeled class of the output neuron in question, e.g. the connection associated with a neuron recently represented by label 1 must be 1.

```
procedure MBabCoNNLearner(E)
begin
  currentAccuracy ← 0,
  previousAccuracy ← 0
  output ← MTluTraining(E)   {output layer with M neurons for a M-class problem}
  currentAccuracy ←  evaluateNetwork(E)
  h ← 0 {hidden neuron index}
  while (currentAccuracy > previousAccuracy) and (currentAccuracy < 1) do
    begin
      highest_wrongly-on_error(E,WrongNeuron,Wrongly-onClass)
      twoClassesE ← createTrainingSet(WrongNeuron,Wrongly-onClass,E)
      h ← h + 1
      hiddenLayer[h] ← bcp(twoClassesE)          {hidden BabCoNN neuron}
      previousAccuracy ←  currentAccuracy
      currentAccuracy ← evaluateNetwork(E)
    end
  if currentAccuracy ≠ 1 then begin
                                 remove(hiddenLayer,h)
                                 h ← h – 1
                               end
end procedure.
```

Fig. 4 Pseudocode of the MBabCoNN learning procedure.

As mentioned before in situations of uncertainty, BabCoNN neurons fire 0; this is convenient in a multiclass situation because it causes no side effects concerning the other patterns that do not belong to either two classes responsible for the hidden neuron creation. After a hidden neuron is added, the whole training set is input to the network grown so far and the classification process is repeated

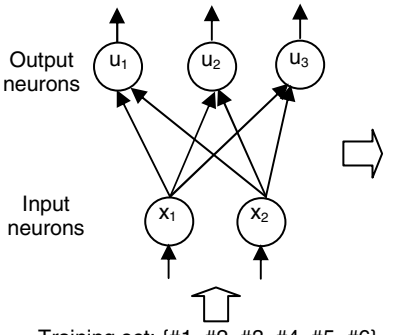
again. Depending on the classification accuracy, new hidden neurons may be added to the network in a similar fashion as the one previously described. If with the addition of a new hidden neuron the accuracy of the network decreases, the new hidden neuron is removed and the learning process ends. The other trivial stopping criteria is the convergence of the network i.e., when the network makes no mistakes).

```
procedure highest_wrongly-on_error(E,WrongNeuron,Wrongly-onClass)
begin
{initializing error matrix}
 for i ← 1 to M do
  for j ← 1 to M do
    outputErr[i,j] ← 0
{collecting errors made by output neurons in training set E={E1,E2,...,EN} }
 for i ← 1 to N do
  begin
    predClass ← MBabCoNN(Ei)
    if predClass ≠ class(Ei)
       then outputErr[predClass,class(Ei)] ← outputErr[predClass,class(Ei)] + 1
  end
{identifying which neuron makes the highest number of wrongly-on errors within a class}
highWrong ← 0
highErr ← 0
highWrongly-onClass ← 0
for i ← 1 to M do
   for j ← 1 to M do
   if outputErr[i,j] > highErr
    then begin
            highErr ← outputErr[i,j]
            highWrong ← i
            highWrongly-onClass ← j
          end
WrongNeuron ← highWrong
Wrongly-onClass ← highWrongly-onClass
end procedure.
```

Fig. 5 Pseudocode for determining the neuron responsible for the highest number of *wrongly-on* misclassifications as well as for the corresponding misclassified class.

4.3 An Example of the MBabCoNN Learning Algorithm

This section shows a simple example of the MBabCoNN learning algorithm according to the pseudocode described in Fig. 4. The example considers a training set with six training patterns identified by numbers 1 to 6 describing three classes identified by numbers 1 to 3. The figure on the left shows a MBabCoNN network after training the output neurons and, on the right, the four misclassifications it makes. The following figures show the evolution of the network implemented by MBabCoNN in the process of correcting the misclassifications.



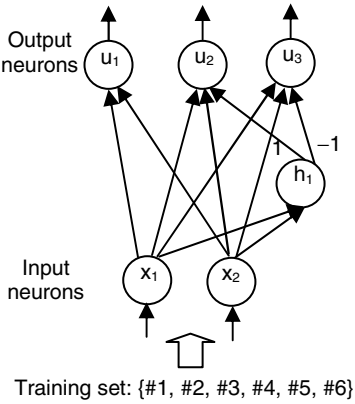
Training set: {#1, #2, #3, #4, #5, #6}

#P	Output neurons u_1	u_2	u_3	Class	
#1	1	–1	–1	1	C
#2	–1	–1	1	2	W
#3	–1	–1	1	2	W
#4	–1	–1	1	1	W
#5	–1	1	–1	3	W
#6	–1	–1	1	3	C

#P: pattern id; C: correctly classified;
W: wrongly classified

Number of *wrongly-on* errors by u_1: 0
Number of *wrongly-on* errors by u_2: 1
Number of *wrongly-on* errors by u_3: 3 (patterns #2, #3 (class 2) and #4 (class 1))
u_3: has the highest number of *wrongly-on* errors within a class (misclassifies #2 and #3 from class 2). A new hidden neuron (h_1) is added to the network and trained with all patterns belonging to class 2 and class 3 i.e., hidden neuron h_1 is trained with E = {#2,#3,#5,#6}.

	class	new class label
#2	2	1
#3	2	1
#5	3	–1
#6	3	–1



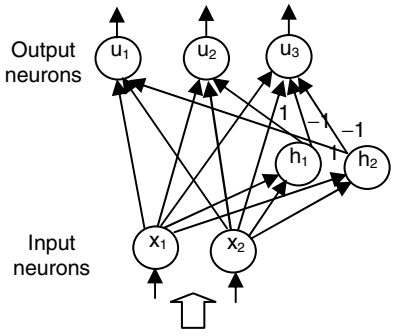
Training set: {#1, #2, #3, #4, #5, #6}

#P	Output neurons u_1	u_2	u_3	Class	
#1	1	–1	–1	1	C
#2	–1	**1**	**–1**	2	**C**
#3	–1	**1**	**–1**	2	**C**
#4	–1	–1	1	1	W
#5	–1	1	–1	3	W
#6	–1	–1	1	3	C

After the addition of h_1, patterns #2 and #3 are correctly classified.

Number of *wrongly-on* errors by u_1: 0
Number of *wrongly-on* errors by u_2: 1
Number of *wrongly-on* errors by u_3: 1 (pattern #4 (class 1))
u_2 and u_3 have the highest number of *wrongly-on* errors within a class. Randomly choose one of them; u_3 for example. A new hidden neuron (h_2) is added to the network and trained with all patterns belonging to class 1 and class 3 i.e., trained with E = {#1,#4,#5,#6}.

	class	new class label
#1	1	1
#4	1	1
#5	3	–1
#6	3	–1



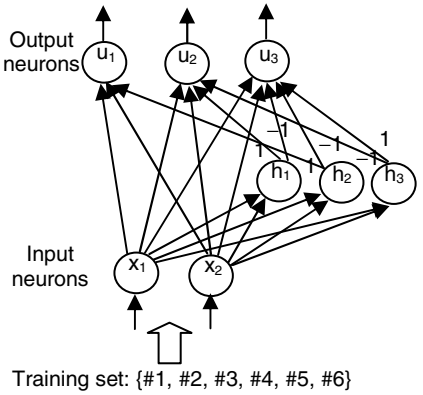
Training set: {#1, #2, #3, #4, #5, #6}

#P	Output neurons u_1	u_2	u_3	Class	
#1	1	−1	−1	1	C
#2	−1	1	−1	2	C
#3	−1	1	−1	2	C
#4	**1**	−1	**−1**	1	**C**
#5	−1	1	−1	3	W
#6	−1	−1	1	3	C

After the addition of h_2, pattern #4 is correctly classified.

Number of *wrongly-on* errors by u_1: 0
Number of *wrongly-on* errors by u_2: 1 (pattern #5 (class 3))
Number of *wrongly-on* errors by u_3: 0
u_2 has the highest number of *wrongly-on* errors within a class. A new hidden neuron (h_3) is added to the network and trained with all patterns belonging to class 3 and class 2 i.e., trained with E = {#5, #6, #2, #3}.

	class	new class label
#5	3	1
#6	3	1
#2	2	−1
#3	2	−1



Training set: {#1, #2, #3, #4, #5, #6}

#P	Output neurons u_1	u_2	u_3	Class	
#1	1	−1	−1	1	C
#2	−1	1	−1	2	C
#3	−1	1	−1	2	C
#4	1	−1	−1	1	C
#5	−1	**−1**	**1**	3	**C**
#6	−1	−1	1	3	C

Network makes no mistakes. Training is finished.

4.4 The MBabCoNN Classifying Algorithm

The MBabCoNN classifying algorithm is described in Fig. 6. For the classification process an output neuron that has any connections to hidden neurons is said to have dependencies.

In the pseudocode of Fig. 6, the procedure *classification( )* approaches the network as constituted by a single output node indexed by *output[ ]* and no hidden neurons; the procedure gives as result the classification of the input pattern by the output node.

The classification process promotes the lack of dependency; if an output neuron fires 1 and has no dependencies then the class given to the pattern being classified is the class the neuron represents. Intuitively, an output neuron that does not create dependencies reflects the fact that it has not been associated with misclassifications during training. This can be an indication that the class represented by this particular neuron is reasonably easy to identify from the others (i.e., is linearly separable from the others). Figure 7 shows an example of this situation.

```
procedure MBabCoNN_classifier(X)
{X: new pattern}
{MBabCoNN network with M output neurons}
begin
 result ← 0, counter ← 0, neuronIndex ← 0
j ← 1
while (j ≤ M) and (counter < 2) do
  begin
    OutputClassification[j] ← classification(output[j],X)  {retrieves 1 or -1}
   if OutputClassification[j] = 1 then
         begin
            counter ← counter + 1
             neuronIndex ← j
         end
   j ←  j + 1
  end
 if ( (counter = 1) and not hasDependencies(output[neuronIndex]) )
  then result ← class(neuronIndex)
  else
   begin
     for j  ← 1 to M do
      begin
       sum ← 0
       for k ← 1 to h do
           begin     {h is the number of  hidden neurons}
            if isConected(k,j)  then   {verifies connection between hidden neuron k and output j}
                  sum ← sum + classification(hiddenLayer[k], X) {BabCoNN-like neuron}
             hiddenClassification[j] ←  sum
          end
      end
    result ← class(greatest(hiddenClassification))
    {returns the class associated with the  index of the greatest value in hiddenClassification}
   end
end  procedure.




end procedure.
```

Fig. 6 Pseudocode of the MBabCoNN procedure for classifying a new pattern.

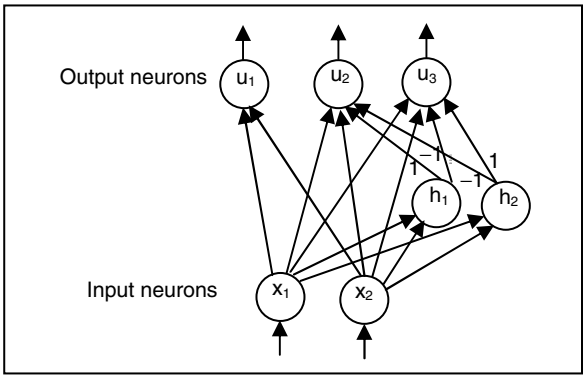


Fig. 7 Two out of three output neurons have dependencies.

In Fig. 7, two of the output neurons, u_2 and u_3 have dependencies (have connections with hidden neurons) and neuron u_1 does not have dependencies. If a new pattern is to be classified, the classification procedure checks if the output given by the node(s) that has (have) no dependencies (in this example, the u_1) is +1; if that is the case the new pattern is assigned the class represented by u_1 otherwise, the classification procedure takes into consideration only the sum of the outputs by the hidden neurons.

In cases where hidden neurons fire value 0, the classification procedure ignores the hidden neurons and takes into account the information given by the output neurons only. If, however, the output neuron that classifies the pattern has dependencies, the output result will be the sum of the outputs of all hidden neurons. If the sum is 0 the output neuron will be in charge of classifying the pattern.

The three output neurons, u_1, u_2 and u_3, in the MBabCoNN network of Fig. 8 have dependencies. Each output node has two connections with the added hidden neurons. A pattern to be classified will result in three outputs, one from each of the three hidden nodes (+1, −1 or 0), which will be multiplied by the connection

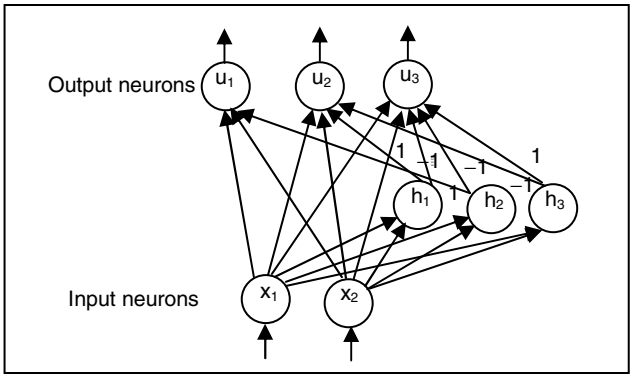


Fig. 8 MBabCoNN network where the three output neurons have dependencies.

weight, producing values +1, −1 or 0. Each output neuron will sum the input received from the hidden neurons and the pattern will be assigned the class represented by the output neuron with the highest score.

5 Experimental Results and Discussion

This section presents and compares the results of using MBabCoNN and the five other multiclass CoNN algorithms previously described, when learning from eight multiclass knowledge domains. Each algorithm was implemented in Java using two different algorithms for training individual TLUs, namely, the PRM and the BCP, identified in tables 2 to 9 by the suffixes P and B added to their names respectively.

Also for comparative purposes, the results of running a multiclass version of PRM and BCP, each implemented in two versions, WTA and independent(I), are presented. The eight knowledge domains used in the experiments have been downloaded from the UCI-Repository [8] and are summarized in Table 1.

Taking into consideration the MBabCoNN proposal (implemented in two versions: MBabCoNNP and MBabCoNNB), the two different versions implemented for each of the five algorithms (MTower, MPyramid, MUpstart, MTiling and MPerceptron-Cascade) and the two different strategies employed for implementing the MPRM and the MBCP, a total of 16 different algorithms have been implemented and evaluated.

Versions MBabCoNNP and MBabCoNNB differ in relation to the algorithm used for training their output neurons, the PRMWTA and the BCPWTA respectively, since both versions use the BCP for training hidden neurons. In the experiments, the accuracy of each neural network is based on the percentage of successful predictions on test sets for each domain. For each of the eight datasets the experiments consisted of performing a ten-fold cross-validation process with each of the 16 algorithms. The results are the average of the ten runs followed by their standard deviation.

Table 1 Domain Specifications

Domain	# PATTERNS	# ATTRIBUTES	# CLASSES
Iris	150	4	3
E. coli	336	7	8
Glass	214	9	6
Balance	625	4	3
Wine	178	13	3
Zoo	101	17	7
Car	1,728	6	4
Image Segmentation	2,310	19	7

Runs with the various learning procedures were carried out on the same training sets and evaluated on the same test sets. The cross-validation folds were the same for all the experiments in each domain. For each domain, each learning procedure was run considering one, ten, a hundred and a thousand iterations; only the best test accuracy among these iterations for each algorithm is presented. All the results obtained with MBabCoNN and the other algorithms (and their variants) are presented in tables 2 to 9, organized by knowledge domain.

The following abbreviations were adopted for presenting the tables: #I: number of iterations, TR training set, TE testing set. The accuracy (Acc) is given as a percentage followed by the standard deviation value. The 'Absolute Best' (AB) column gives the best performance of the learning procedure (in TE) over the ten runs and the 'Absolute Worst' (AW) column gives the worst performance of the learning procedure (in TE) over the ten runs; #HN represents the number of hidden nodes; AB(HN) gives the smallest number of hidden nodes created and AW(HN) gives the highest number of hidden nodes created.

Obviously the PRMWTA, PRMI, BCPWTA and BCPI do not have values for #HN, AB(HN) and AW(HN) because the networks they create only have input and output layers.

In relation to the results obtained in the experiments shown in tables 2 to 9, it can be said that as far as accuracy in test sets is concerned, MBabCoNNP has shown the best performance in four out of eight domains, namely the Iris, E. Coli, Wine and Zoo. In the Balance domain, although its result is very close to the best

Table 2 Iris

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^3	98.7~0.4	98.0~3.2	3.1~0.3	100.0	93.3	3.0	4.0
MBabCoNNB	10^2	94.2~3.1	94.0~4.9	4.9~0.6	100.0	86.7	6.0	4.0
PRMWTA	10^2	98.7~0.4	95.3~8.9	–	100.0	7.3.3	–	–
PRMI	10^2	89.3~2.6	86.7~18.1	–	100.0	46.7	–	–
BCPWTA	10^2	87.9~1.4	84.0~13.4	–	100.0	53.3	–	–
BCPI	1	85.0~7.3	72.7~41.3	–	100.0	6.7	–	–
MTowerP	10^2	98.9~0.4	96.7~6.5	3.6~1.3	100.0	80.0	3.0	6.0
MTowerB	10^2	87.8~1.6	83.3~14.5	3.3~0.9	100.0	53.3	3.0	6.0
MPyramidP	10^2	98.8~0.4	96.0~8.4	3.3~0.9	100.0	73.3	3.0	6.0
MPyramidB	10^2	88.3~1.4	82.7~13.8	3.6~1.3	100.0	53.3	3.0	6.0
MUpstartP	10^2	98.9~0.4	93.3~12.9	3.3~0.0	100.0	60.0	3.0	3.0
MUpstartB	10^2	88.6~1.6	80.7~13.9	3.6~0.5	100.0	53.3	3.0	4.0
MTilingP	10	98.2~0.8	95.3~8.9	3.0~0.0	100.0	73.3	3.0	3.0
MTilingB	10^2	89.6~4.7	84.0~14.5	7.0~8.8	100.0	53.3	3.0	28.0
MPCascadeP	10^2	98.8~0.4	95.3~8.9	3.0~0.0	100.0	73.3	3.0	3.0
MPCasdadeB	10^2	88.5~1.4	80.7~13.5	3.3~0.5	100.0	53.3	3.0	4.0

Table 3 E. Coli

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^2	90.6~0.6	83.9~5.3	8.2~0.6	91.2	75.8	8.0	9.0
MBabCoNNB	10^2	85.1~1.5	81.0~5.2	9.1~0.7	88.2	73.5	10.0	8.0
PRMWTA	10^2	90.8~1.3	77.8~18.9	–	100.0	52.9	–	–
PRMI	10^2	87.2~2.6	73.8~24.0	–	100.0	42.4	–	–
BCPWTA	10^2	76.5~3.0	69.1~15.0	–	97.1	42.4	–	–
BCPI	10	85.1~3.3	72.0~28.2	–	100.0	27.3	–	–
MTowerP	10	90.2~1.5	78.4~22.2	27.8~14.1	100.0	30.3	6.0	56.0
MTowerB	10^2	76.6~2.9	69.2~14.4	9.2~3.2	97.1	48.5	8.0	16.0
MPyramidP	10	90.1~1.3	79.6~18.3	29.1~10.3	100.0	42.4	14.0	48.0
MPyramidB	10^2	76.6~2.8	69.5~16.2	8.4~2.1	97.1	36.4	6.0	14.0
MUpstartP	10^2	90.7~1.6	76.9~19.7	8.1~1.3	100.0	50.0	6.0	10.0
MUpstartB	10^2	80.5~2.8	75.2~10.4	8.9~1.3	97.1	60.7	6.0	11.0
MTilingP	10	87.9~2.2	76.3~20.7	7.7~0.7	100.0	38.2	6.0	8.0
MTilingB	10^2	76.3~3.3	67.3~18.0	25.3~55.4	97.1	33.3	6.0	183.0
MPCascadeP	10	88.8~1.2	82.9~16.2	8.6~1.3	100.0	57.8	8.0	11.0
MPCasdadeB	10^2	79.6~2.7	70.1~15.3	9.0~1.7	97.0	42.4	6.0	12.0

Table 4 Glass

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^3	64.7~2.5	60.3~12.7	7.5~0.7	68.4	60.6	85.7	42.9
MBabCoNNB	10^3	63.2~2.2	56.1~8.1	7.4~0.5	66.8	60.9	66.7	40.9
PRMWTA	10^3	55.7~1.4	48.1~14.3	_	57.5	52.3	71.4	23.8
PRMI	10^3	53.9~2.6	46.8~14.0	_	57.3	50.0	81.0	28.6
BCPWTA	10^2	54.3~2.7	49.6~10.2	_	59.6	48.7	66.7	36.4
BCPI	10^3	58.8~5.2	54.2~5.8	_	66.7	51.6	66.7	47.6
MTowerP	10^3	61.2~2.5	56.1~10.8	21.6~8.6	65.8	58.0	71.4	33.3
MTowerB	10^3	56.1~1.9	50.6~6.8	18.6~10.4	60.4	53.4	61.9	36.4
MPyramidP	10^2	64.9~5.0	56.3~13.9	33.0~17.7	71.0	56.5	77.3	28.6
MPyramidB	10^2	55.1~2.3	51.5~11.0	17.4~4.4	59.1	52.1	66.7	36.4
MUpstartP	10^3	72.2~1.5	63.6~6.8	8.3~0.9	74.6	69.3	76.2	52.4
MUpstartB	10^2	64.3~4.7	54.7~8.4	7.9~1.0	68.2	52.8	66.7	42.9
MTilingP	10^2	79.9~16.5	55.9~15.2	81.5~63.1	92.7	54.9	72.7	23.8
MTilingB	10^3	54.6~1.5	50.0~8.2	6.0~0.0	57.5	52.1	61.9	38.1
MPCascadeP	10^3	71.4~1.8	62.7~10.4	7.6~1.2	73.6	68.2	81.0	42.9
MPCasdadeB	10^2	57.6~5.2	54.4~13.4	7.2~1.2	65.8	48.7	76.2	36.4

Table 5 Balance

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10	92.1~0.8	91.4~2.3	3.5~0.7	93.7	87.1	3.0	5.0
MBabCoNNB	10	92.1~0.9	89.3~2.5	5.3~0.5	93.5	85.5	6.0	5.0
PRMWTA	10^2	92.2~0.5	90.1~3.7	–	96.8	84.1	–	–
PRMI	10	89.1~1.6	89.3~4.8	–	98.4	84.1	–	–
BCPWTA	10^2	80.1~4.4	77.9~9.9	–	91.9	65.1	–	–
BCPI	10	89.5~1.7	88.0~3.5	–	93.5	82.3	–	–
MTowerP	10	94.8~1.1	90.6~6.2	20.4~5.8	98.4	79.4	12.0	30.0
MTowerB	10^2	83.6~4.8	80.8~8.0	9.0~3.5	91.9	65.1	6.0	15.0
MPyramidP	10	95.1~0.9	90.1~6.3	24.0~6.2	96.8	76.2	15.0	33.0
MPyramidB	10^2	83.3~4.1	83.1~5.8	6.2~2.2	93.5	76.2	3.0	9.0
MUpstartP	10	91.8~0.4	91.2~4.7	3.4~0.9	98.4	85.5	3.0	6.0
MUpstartB	10^2	82.1~4.9	81.1~6.9	4.0~0.8	91.9	69.8	3.0	5.0
MTilingP	10^2	95.5~2.9	92.3~3.3	28.1~21.8	96.8	88.9	3.0	49.0
MTilingB	10^2	79.8~4.5	76.4~8.4	3.0~0.0	91.9	66.7	3.0	3.0
MPCascadeP	10^2	92.1~0.6	90.0~4.5	3.2~0.4	98.4	84.1	3.0	4.0
MPCasdadeB	10^2	81.6~4.7	78.6~9.9	4.0~0.8	91.9	66.7	3.0	5.0

Table 6 Wine

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^3	94.2~0.9	92.7~5.5	3.3~0.5	95.0	93.1	100.0	82.4
MBabCoNNB	10^2	76.2~2.2	75.8~8.8	4.4~0.5	81.9	73.8	88.9	58.8
PRMWTA	10^3	85.8~3.3	81.9~10.4	_	90.1	78.9	100.0	64.7
PRMI	10^3	92.0~1.4	88.3~8.9	_	94.4	90.0	100.0	66.7
BCPWTA	10^3	74.0~1.4	70.8~7.1	_	75.6	70.8	82.4	61.1
BCPI	10	73.7~1.7	71.9~7.3	_	75.8	70.0	77.8	58.8
MTowerP	10^3	86.5~3.5	83.1~13.5	4.2~2.1	90.0	81.2	100.0	61.1
MTowerB	10^2	73.6~0.9	69.1~7.6	4.2~1.5	74.5	71.9	83.3	61.1
MPyramidP	10^3	87.3~2.7	81.5~17.1	8.1~4.9	90.7	83.1	100.0	50.0
MPyramidB	10^3	73.6~0.9	69.1~7.6	4.2~15	74.5	71.9	83.3	61.1
MUpstartP	10^3	94.5~0.5	91.6~6.0	3.1~0.3	95.6	93.8	100.0	83.3
MUpstartB	10^2	74.1~1.6	71.4~11.3	3.2~0.4	76.2	70.8	94.1	55.6
MTilingP	10^2	86.7~5.0	82.5~11.2	13.1~31.9	99.4	81.9	94.4	61.1
MTilingB	10^2	73.8~2.1	74.8~8.7	3.0~0.0	76.2	69.6	88.9	61.1
MPCascadeP	10^3	92.1~0.7	84.9~9.0	3.1~0.3	93.1	91.2	94.1	66.7
MPCasdadeB	10	75.7~1.2	74.1~9.2	3.2~0.4	77.5	73.3	88.9	61.1

Table 7 Zoo

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^2	100.0~0.0	97.0~4.8	7.0~0.0	100.0	100.0	100.0	90.0
MBabCoNNB	10^2	89.5~5.1	85.1~12.7	9.3~0.9	95.6	77.8	100.0	60.0
PRMWTA	10^2	100.0~0.0	95.2~6.7	–	100.0	100.0	100.0	81.8
PRMI	10^2	100.0~0.0	95.0~7.1	–	100.0	100.0	100.0	80.0
BCPWTA	10^3	78.4~7.5	71.3~16.6	–	89.0	65.6	90.0	40.0
BCPI	10	64.9~16.0	65.5~16.1	–	94.5	36.3	90.0	40.0
MTowerP	10^2	100.0~0.0	95.0~7.1	7.0~0.0	100.0	100.0	100.0	80.0
MTowerB	10^2	79.0~5.3	70.4~7.7	9.1~3.4	86.8	72.5	80.0	60.0
MPyramidP	10^2	100.0~0.0	95.0~8.5	7.0~0.0	100.0	100.0	100.0	80.0
MPyramidB	10^2	78.2~7.9	74.3~15.0	9.8~4.9	86.7	64.8	100.0	50.0
MUpstartP	10^3	100.0~0.0	96.0~7.0	7.0~0.0	100.0	100.0	100.0	80.0
MUpstartB	10^2	81.0~6.6	74.4~13.1	7.9~0.7	90.1	70.3	90.0	50.0
MTilingP	10^3	100.0~0.0	97.0~4.8	7.0~0.0	100.0	100.0	100.0	90.0
MTilingB	10^3	78.5~7.9	72.4~12.8	13.3~10.2	90.1	65.9	90.0	50.0
MPCascadeP	10^3	100.0~0.0	95.1~7.0	7.0~0.0	100.0	100.0	100.0	80.0
MPCasdadeB	10^2	77.8~6.2	72.4~15.2	7.4~0.7	85.7	68.1	100.0	50.0

Table 8 Car

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^3	81.2~0.7	80.1~2.8	4.7~0.5	82.8	80.4	83.8	76.3
MBabCoNNB	10^2	78.3~3.6	77.5~4.4	5.8~0.9	80.5	68.2	84.4	70.5
PRMWTA	10^2	80.5~0.5	79.7~1.7	–	81.4	79.9	82.7	78.0
PRMI	10^2	79.1~0.5	78.9~4.0	–	80.2	78.3	83.8	72.1
BCPWTA	10^3	68.7~0.6	68.5~3.1	–	69.4	67.3	73.8	63.0
BCPI	10^2	77.3~1.1	76.3~2.7	–	79.4	75.8	81.5	73.4
MTowerP	10^2	82.7~1.2	81.5~2.4	26.8~18.4	85.1	81.6	86.1	78.5
MTowerB	10^3	73.8~2.1	73.3~3.7	5.2~2.7	75.4	68.2	78.6	65.7
MPyramidP	10	83.6~1.3	80.4~3.8	32.8~14.2	85.7	81.1	86.7	73.8
MPyramidB	10^2	68.6~1.2	68.5~4.1	5.2~2.7	71.2	67.2	78.0	64.7
MUpstartP	10^2	81.9~1.7	80.8~2.3	5.3~1.6	85.3	80.3	85.5	77.5
MUpstartB	10	75.3~0.3	74.8~3.3	4.3~0.5	76.1	74.9	79.2	69.4
MTilingP	10^2	89.2~3.0	83.0~3.9	88.1~30.3	91.1	80.6	86.7	73.4
MTilingB	10	74.9~0.5	74.7~2.3	4.0~0.0	75.3	73.6	79.8	72.1
MPCascadeP	10^3	82.8~1.4	81.2~4.4	6.3~1.7	84.3	80.2	87.9	74.4
MPCasdadeB	10^3	74.8~1.7	73.8~4.9	4.7~0.8	75.8	70.1	79.2	62.2

Table 9 Image Segmentation

Algorithm	#I	Acc(TR)	Acc(TE)	#HN	AB(TE)	AW(TE)	AB(HN)	AW(HN)
MBabCoNNP	10^3	92.2~1.6	83.3~7.2	7.9~0.7	95.2	89.4	95.2	71.4
MBabCoNNB	10^3	74.0~4.7	70.0~9.5	8.8~0.9	81.5	68.8	85.7	52.4
PRMWTA	10^3	91.2~0.9	83.8~10.6	–	92.6	89.4	100.0	66.7
PRMI	10^3	88.9~1.5	83.8~6.4	–	91.5	86.8	90.5	71.4
BCPWTA	10^2	62.2~2.3	60.5~6.8	–	65.6	57.1	71.4	52.4
BCPI	10^2	67.6~5.9	63.8~15.9	–	78.8	60.3	81.0	33.3
MTowerP	10^2	91.7~1.2	82.4~9.0	14.0~8.1	93.1	88.9	95.2	71.4
MTowerB	10^2	67.9~3.7	63.8~14.1	11.9~4.7	72.0	61.9	85.7	42.9
MPyramidP	10^3	92.0~1.1	81.4~4.7	11.2~4.9	93.1	89.9	85.7	71.4
MPyramidB	10^3	61.3~3.0	60.5~8.7	9.1~3.4	68.3	57.7	81.0	52.4
MUpstartP	10^3	94.8~0.7	85.2~10.9	7.1~0.3	95.8	93.7	100.0	66.7
MUpstartB	10^3	64.7~2.9	59.5~8.5	7.3~0.7	68.8	58.2	71.4	47.6
MTilingP	10^2	93.4~4.0	82.4~7.1	35.5~65.9	100.0	89.9	90.5	71.4
MTilingB	10^2	64.7~5.0	63.8~8.8	7.0~0.0	70.9	54.0	76.2	52.4
MPCascadeP	10^2	88.8~1.2	86.2~6.1	7.3~0.5	90.5	87.3	90.5	76.2
MPCasdadeB	10^3	66.7~4.0	64.8~10.3	7.4~0.7	70.4	57.1	81.0	47.6

result (obtained with MTilingP), it is worth noting that MTilingP created 28.1 hidden neurons on average while MBabCoNN created only 3.5. A similar situation occurred in the Car domain.

All the algorithms with a performance higher than 80% induced networks bigger than the network induced by MBabCoNNP, especially the MTilingP, the MPyramid and the MTowerP, although the accuracy values of the three were very close to those of MBabCoNN. In the Glass domain, MBabCoNNP is ranked third considering only accuracy; however, when taking into account the standard deviation as well, it can be said that the MBabCoNNP and MPCascadeP (second position in the rank) are even. In the last domain, Image Segmentation, MBabCoNNP accuracy was average while the best performance was obtained with the MPCascadeP.

In relation to the versions that used BCPWTA for training the output neuron, MBabCoNNB outperformed all the other algorithms in the eight domains. The results however were inferior to those obtained using the PRMWTA for training the output nodes. This fact is due to the particular characteristics of the two training approaches; in general the BCPWTA is not a good match for the PRMWTA. Future work concerning BCPWTA needs to be done in order for this algorithm to be considered an option for network construction.

Now, considering the PRMWTA versions of all algorithms, it is easy to see that the test accuracies are more standardized. In order to have a clearer view of the algorithm performances, Table 10 presents the values for the average ranks considering the test accuracy for all PRMWTA based CoNN algorithms. In

the table the results are ranked in ascending order of accuracy. Ties in accuracy were sorted out by averaging the corresponding ranks.

According to Demšar [26], average ranks provide a fair comparison of the algorithms. Taking into account the values obtained with the average ranks it can be said that, as far as the eight datasets are concerned, MBabCoNN is the best choice among the six algorithms. As can be seen in Table 10, MBabCoNNP obtained the smallest value, followed by MPCascadeP and MUpstartP. MBabCoNNP was ranked last in only one domain (Car); in the Car domain, however, the test accuracies among the six algorithms were very close, i.e. the maximum difference was about 3.0%.

It is worth noticing that the three algorithms ranked first in the average rankings, construct the network by first adding the output neurons and then starting to correct their misclassifications by adding two-class hidden neurons. The good performance may be used to corroborate the efficiency of this technique. Based on the empirical results obtained, it can be said that the MBabCoNN algorithm is a good choice among the multiclass CoNN algorithms available.

Table 10 Average rank over PRMWTA based algorithms concerning test accuracy

Domain	MBabCoNNP	MTowerP	MPyramidP	MUpstartP	MTilingP	MPCascadeP
Iris	98.0~3.2 (**1**)	96.7~6.5(**2**)	96.0~8.4(**3**)	93.3~12.9(**6**)	95.3~8.9(**4.5**)	95.3~8.9(**4.5**)
EColi	83.9~5.3(**1**)	78.4~22.2(**4**)	79.6~18.3(**3**)	76.9~19.7(**5**)	76.3~20.7(**6**)	82.9~16.2(**2**)
Glass	60.3~12.7(**3**)	56.1~10.8(**5**)	56.3~13.9(**4**)	63.6~6.8(**1**)	55.9~15.2(**6**)	62.7~10.4(**2**)
Balance	91.4~2.3(**2**)	90.6~6.2(**4**)	90.1~6.3(**5**)	91.2~4.7(**3**)	92.3~3.3(**1**)	90.0~4.5(**6**)
Wine	92.7~5.5(**1**)	83.1~13.5(**4**)	81.5~17.1(**6**)	91.6~6.0(**2**)	82.5~11.2(**5**)	91.0~6.1(**3**)
Zoo	97.0~4.8(**1.5**)	95.0~7.1(**5.5**)	95.0~8.5(**5.5**)	96.0~7.0(**3**)	97.0~4.8(**1.5**)	95.1~7.0(**4**)
Car	80.1~2.8(**6**)	81.5~2.4(**2**)	80.4~3.8(**5**)	80.8~2.3(**4**)	83.0~3.9(**1**)	81.2~4.4(**3**)
Image	83.3~7.2(**3**)	82.4~9.0(**4.5**)	81.4~4.7(**6**)	85.2~10.9(**2**)	82.4~7.1(**4.5**)	86.2~6.1(**1**)
Average Rank	2.312	3.875	4.687	3.25	3.687	3.187

5 Conclusions

This chapter proposes the multiclass version, MBabCoNN, of a recently proposed constructive neural network algorithm named BabCoNN, which is based on the geometric concept of convex hull and uses the BCP algorithm for training individual TLUs added to the network during learning. The chapter presents the accuracy results of learning experiments conducted in eight multiclass knowledge domains, using the MBabCoNN implemented in two different versions: MBabCoNNP and MBabCoNNB, *versus* five well-known multiclass algorithms (each implemented in two versions as well). Both versions of the MBabCoNN use the BCP for training the hidden neurons and differ from each other in relation to the algorithm used for training their output neurons (PRMWTA and BCPWTA respectively).

As far as results in eight knowledge domains are concerned, it can (easily) be observed that all algorithms performed better when using PRMWTA for training the output neurons. This may occur because BCPWTA is not a good strategy for

training M (>2) classes. Now considering the PRMWTA versions, it can be said the MBabCoNNP version has shown superior average performance in relation to both accuracy in test sets and the size of the induced neural network. This work had established MBabCoNN as a good option among other CoNNs for multiclass domains.

Acknowledgments. To CAPES and FAPESP for funding the work of the first author and for the project financial help granted to the second author, and to Leonie C. Pearson for proofreading the first draft of this chapter.

References

[1] Mitchell, T.M.: Machine learning. McGraw-Hill, USA (1997)
[2] Duda, R.O., Hart, P.E., Stork, D.G.: Pattern classification. John Wiley & Sons, USA (2001)
[3] Nilsson, N.J.: Learning machines. McGrall-Hill Systems Science Series, USA (1965)
[4] Elizondo, D.: The linear separability problem: some testing methods. IEEE Transactions on Neural Networks 17(2), 330–344 (2006)
[5] Bertini Jr., J.R., Nicoletti, M.C.: A constructive neural network algorithm based on the geometric concept of barycenter of convex hull. In: Rutkowski, L., Tadeusiewiza, R., Zadeh, L.A., Zurada, J. (eds.) Computational Intelligence: Methods and Applications, 1st edn., vol. 1, pp. 1–12. Academic Publishing House EXIT, Warsaw (2008)
[6] Bertini Jr., J.R., Nicoletti, M.C.: MBabCoNN - a multiclass version of a constructive neural network algorithm based on linear separability and convex hull. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part II. LNCS, vol. 5164, pp. 723–733. Springer, Heidelberg (2008)
[7] Poulard, H.: Barycentric correction procedure: A fast method for learning threshold unit. In: WCNN 1995, Washington, DC, vol. 1, pp. 710–713 (1995)
[8] Asuncion, A., Newman, D.J.: UCI machine learning repository. University of California, School of Information and Computer Science, Irvine (2007), `http://www.ics.uci.edu/~mlearn/MLRepository.html`
[9] Parekh, R.G.: Constructive learning: inducing grammars and neural networks. Ph.D. Dissertation, Iowa State University, Ames, Iowa (1998)
[10] Nicoletti, M.C., Bertini Jr., J.R.: An empirical evaluation of constructive neural network algorithms in classification tasks. Int. Journal of Innovative Computing and Applications (IJICA) 1, 2–13 (2007)
[11] Gallant, S.I.: Neural network learning & expert systems. The MIT Press, Cambridge (1994)
[12] Mézard, M., Nadal, J.: Learning feedforward networks: the tiling algorithm. J. Phys. A: Math. Gen. 22, 2191–2203 (1989)
[13] Frean, M.: The upstart algorithm: a method for constructing and training feedforward neural networks. Neural Computation 2, 198–209 (1990)
[14] Burgess, N.: A constructive algorithm that converges for real-valued input patterns. International Journal of Neural Systems 5(1), 59–66 (1994)
[15] Amaldi, E., Guenin, B.: Two constructive methods for designing compact feedforward networks of threshold units. International Journal of Neural System 8(5), 629–645 (1997)
[16] Poulard, H., Labreche, S.: A new threshold unit learning algorithm, Technical Report 95504, LAAS (December 1995)
[17] Poulard, H., Estèves, D.: A convergence theorem for barycentric correction procedure, Technical Report 95180, LAAS-CNRS, Toulouse (1995)

[18] Bertini Jr., J.R., Nicoletti, M.C., Hruschka Jr., E.R.: A comparative evaluation of constructive neural networks methods using PRM and BCP as TLU training algorithms. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, pp. 3497–3502. IEEE Press, Los Alamitos (2006)
[19] Bertini Jr., J.R., Nicoletti, M.C., Hruschka Jr., E.R., Ramer, A.: Two variants of the constructive neural network Tiling algorithm. In: Proceedings of The Sixth International Conference on Hybrid Intelligent Systems (HIS 2006), pp. 49–54. IEEE Computer Society, Washington (2006)
[20] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational geometry: algorithms and applications, 2nd edn. Springer, Berlin (2000)
[21] Parekh, R.G., Yang, J., Honavar, V.: Constructive neural network learning algorithms for multi-category real-valued pattern classification. TR ISU-CS-TR97-06, Iowa State University, IA (1997)
[22] Parekh, R.G., Yang, J., Honavar, V.: Constructive neural network learning algorithm for multi-category classification. TR ISU-CS-TR95-15a, Iowa State University, IA (1995)
[23] Parekh, R.G., Yang, J., Honavar, V.: MUpstart – a constructive neural network learning algorithm for multi-category pattern classification. In: ICNN 1997, vol. 3, pp. 1920–1924 (1997)
[24] Yang, J., Parekh, R.G., Honavar, V.: MTiling – a constructive network learning algorithm for multi-category pattern classification. In: Proc. of the World Congress on Neural Networks, pp. 182–187 (1996)
[25] Fahlman, S., Lebiere, C.: The cascade correlation architecture. In: Advances in Neural Information Processing Systems, vol. 2, pp. 524–532. Morgan Kaufmann, San Mateo (1990)
[26] Demšar, J.: Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research 7, 1–30 (2006)