Constructive Training of Probabilistic Neural Networks

Michael R. Berthold

Department of Computer Design and Fault Tolerance (Prof. D. Schmid), University of Karlsruhe, P.O. Box 6980, 76128 Karlsruhe, Germany.
E–mail: Michael.Berthold@informatik.uni-karlsruhe.de

Jay Diamond

Intel Corporation, 2200 Mission College Blvd, MS: RN5-19, Santa Clara, CA 95052-8119, USA.
E–mail: Jay_M_Diamond@ccm.sc.intel.com

Received 10 January 1997; accepted 23 August 1997

Abstract

This paper presents an easy to use, constructive training algorithm for Probabilistic Neural Networks, a special type of Radial Basis Function Networks. In contrast to other algorithms, predefinition of the network topology is not required. The proposed algorithm introduces new hidden units whenever necessary and adjusts the shape of already existing units individually to minimize the risk of misclassification. This leads to smaller networks compared to classical PNNs and therefore enables the use of large datasets. Using eight classification benchmarks from the StatLog project, the new algorithm is compared to other state of the art classification methods. It is demonstrated that the proposed algorithm generates Probabilistic Neural Networks that achieve a comparable classification performance on these datasets. Only two rather uncritical parameters are required to be adjusted manually and there is no danger of overtraining — the algorithm clearly indicates the end of training. In addition, the networks generated are small due to the lack of redundant neurons in the hidden layer.

Keywords: Probabilistic Neural Networks, Pattern Recognition, Constructive Training, Dynamic Decay Adjustment.

1 Introduction

Automatic Pattern Recognition has gained considerable interest in the past few years. Applications include, for example, quality control for industrial products, automatic recognition of faces, signatures or street signs and much

more. One focus of attention are procedures that receive preprocessed data as input (e.g. features extracted from pictures or Fourier transformed pieces of sound) and produce class information as output (e.g. good/defect). Usually the underlying decision process is unknown and the recognizer must be constructed using sample data that was classified by hand. Depending on the type and amount of knowledge about the underlying decision process this training can be done in different ways.

When information about the type of the underlying decision process is known a priori, it is sometimes sufficient to specify the structure of the classifier and simply adjust some of its parameters during training. This could be the slope and bias of a linear decision boundary, or the set of parameters that specify a fixed number of rules. For these approaches a certain amount of knowledge about the underlying process is required which is sometimes difficult to acquire for practical applications. Therefore, techniques for dynamically constructing flexible classifiers without requiring a priori knowledge continue to garner considerable interest. Automatic Rule Learning systems find a flexible set of rules that try to describe the training data (for a few examples see [13,22] and [15,16]). Other approaches try to find more hierarchical structures, the most prominent example probably being Quinlan's c4.5 [9], an algorithm which builds decision trees from examples. In most cases, the techniques used to model the data offer insight into the classification process. The power of expression, however, is mostly limited to simple geometrical structures. More complex structures have to be modeled using a large number of simpler elements, which makes interpretation complicated, if not impossible.

One way to build classifiers that offer greater flexibility is to use Artificial Neural Networks. Here a combination of simple, structurally identical computation devices build up a complex classification structure. Unfortunately the classical Multi Layer Perceptron [12] does not offer explanations for the resulting classification which poses severe problems, especially for industrial and security critical applications. Approaches to extract rules from Multi Layer Perceptrons have been proposed (see for example [21]), but are usually not applicable to large networks and are time consuming in practice. Other neural network models that offer ways to interpret their behavior have been proposed, mostly based on networks of locally–tuned processing units, often called Radial Basis Function Networks (RBF) [7]. One special type of these networks with a more statistical origin, the Probabilistic Neural Network (PNN), was proposed by Specht [17]. The PNN consists of one layer of units with a local, Gaussian activation function and models the probability distribution of each class through a combination (or mixture) of these Gaussians. It has been shown [19] that PNNs offer superior performance on real–world benchmark datasets. The classical PNN is similar to an "intelligent memory" since each training pattern is stored as one unit of the layer of Gaussians. Algorithms that train PNNs are therefore infeasible for large datasets because the resulting network con-

tains as many neurons as there are patterns in the training dataset. Newer algorithms that attempt to reduce the network's size unfortunately require an a priori defined architecture, i.e. the number of used Gaussians must be specified before actual training can take place [20].

The Dynamic Decay Adjustment algorithm (DDA, see [3]) presented in this paper allows the automatic construction of PNNs from even very large datasets. The PNN is dynamically constructed during training and the number of required hidden units is optimized automatically. In addition the region of influence for each Gaussian is computed based on information about neighbors. This technique increases the recognition accuracy in areas of conflict. In contrast to the often used "partition of unity" normalization, a modified normalization method is proposed that allows the approximation of class posteriori probabilities together with an additional "don't know"–probability.

In the next section, PNNs are explained in more detail. In section 3, the new training algorithm is presented, together with some analysis of the used parameters and the modified normalization. Finally, section 4 presents results from both synthetic and real–world datasets.

2 Probabilistic Neural Networks

The Probabilistic Neural Network was introduced in 1990 by Specht [17] and puts the statistical kernel estimator [8] into the framework of Radial Basis Function Networks. PNNs have gained interest because they offer a way to interpret the network's structure in the form of a probability density function and their performance is often superior to other state–of–the–art classifiers [19]. In addition, most training methods for PNNs are easy to use. In contrast to classical RBFs, PNNs are only used for classification and they compute conditional class probabilities $p(\text{class } k|\vec{x})$ for each of c classes. The structure of a PNN is shown in Figure 1. Similar to RBFs, PNNs receive n–dimensional feature vectors $\vec{x} = (x_1, \cdots, x_n)$ as input. This input vector is applied to the input neurons x_i $(1 \leq i \leq n)$ and is passed to the neurons in the first hidden layer. Here m_k Gaussians $\mathcal{N}(\vec{\mu}_j^k, \Sigma_j^k)$ are computed for each class k $(1 \leq k \leq c)$:

$$p_j^k(\vec{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma_j^k|^{-1/2}} \exp\left\{-\frac{1}{2}(\vec{x} - \vec{\mu}_j^k)^{\mathsf{T}}(\Sigma_j^k)^{-1}(\vec{x} - \vec{\mu}_j^k)\right\} \quad (1)$$

where $\vec{\mu}_j^k$ denotes the mean of the distribution and Σ_j^k indicates its covariance matrix. Σ_j^k in this case is a positive definite matrix; that is, all eigenvalues are positive. For each class k, therefore, m_k multivariate distributions exist. The second hidden layer computes the approximation of the class probability

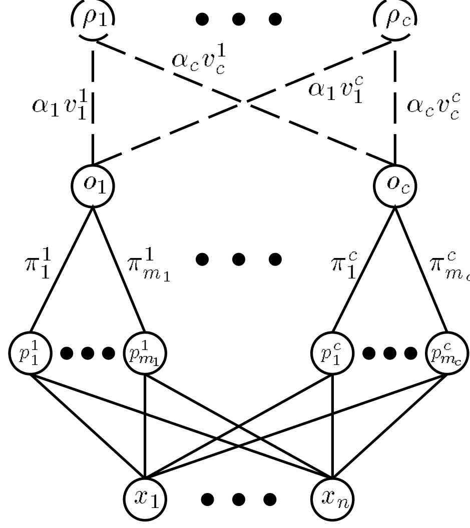

Fig. 1. A typical Probabilistic Neural Net.

functions through a combination of these multivariate densities[1]:

$$o_k(\vec{x}) = \sum_{j=1}^{m_k} \pi_j^k p_j^k(\vec{x}). \tag{2}$$

where π_j^k represents the within–class mixing proportion. The π_j^k are non negative and hold:

$$\sum_{j=1}^{m_k} \pi_j^k = 1 \quad , \quad k = 1, \ldots, c. \tag{3}$$

If there exists a risk function that assigns cost v_l^k to a decision for class k in the case of the pattern $\vec{x}$ actually belonging to class l, a third layer can be used which computes the decision risk:

$$\rho_k(\vec{x}) = \sum_{l=1}^{c} v_l^k \alpha_l o_l(\vec{x}) \tag{4}$$

Here α_l indicates the a priori probability of class l. Using a PNN for a risk–based decision, class l with minimum risk ρ_l would be chosen:

$$l = \mathrm{argmin}_{1 \leq k \leq c}\{\rho_k(\vec{x})\}. \tag{5}$$

[1] This is the main difference between classical RBFs and PNNs. The first hidden layer is not fully connected to the next layer because each neuron of the first layer is already associated with one specific class and therefore only connected to the corresponding neuron in the second hidden layer.

Training of PNNs can be done in a number of ways. Specht [17] proposes the introduction of one neuron for each training pattern, and restricts Σ to one global and scalar smoothing parameter σ. The resulting density function is a sum of homoscedastic Gaussian; "homoscedastic" because only one global smoothing parameter is used. This approach is of course not feasible for large datasets. In addition the adjustment of the smoothing parameter σ has to be done carefully using some validation dataset. Small changes of σ influence the network's performance heavily.

In [18] an extension to this method is proposed that uses a diagonal matrix Σ instead of the scalar σ and iteratively adjusts the diagonal entries of Σ depending on the change in the error. This results in an adaptive normalization of the input space but the adaptation is very time consuming and the problem of the potentially large network size remains.

Other approaches that are able to deal with larger training sets predefine the topology of the network and only adjust the remaining network parameters (Σ_j^k and π_j^k) during training. All of the approaches focus on a homoscedastic network; that is, use only one global covariance matrix Σ. Streit and Luginbuhl [20] propose predefining the number of neurons for each class and then adjusting the parameters using a maximum likelihood training method. Using a global Σ the training data of all classes can be used to adjust this matrix, making the approach feasible for smaller datasets as well.

All these training algorithms rely either on a predefined network topology, or are not feasible for large datasets. Parameters that must be adjusted carefully have a dramatic influence on the final classification performance. In this paper, an algorithm is proposed that constructs the topology of the network during training and thus determines the number of required neurons automatically. In addition, the shape of each Gaussian is adjusted individually through a local, scalar smoothing parameter σ_i resulting in the construction of heteroscedastic PNNs. The algorithm is easy to use and offers fast training with only two user–controllable but uncritical parameters. The resulting Probabilistic Neural Networks offer a comparable performance to classical PNNs but with a reduced network size which makes them also applicable to large datasets.

3 The Dynamic Decay Adjustment Algorithm

The algorithm presented in this paper is based on the RCE–algorithm [4,10] and introduces the idea of distinguishing between matching and conflicting neighbors in an area of conflict. Two thresholds θ^+ and θ^- are used during training as illustrated in Figure 2. θ^+ determines the minimum correct–classification probability for training patterns of the correct class. In contrast

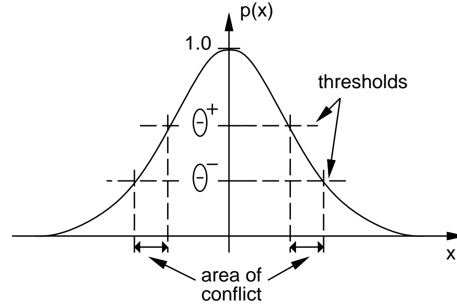

Fig. 2. The two thresholds used by the DDA algorithm.

θ^- is used to avoid misclassifications; that is, the probability for an incorrect class for each training pattern is less than or equal to θ^-. In a geometrical analogy, this leads to an area of conflict where neither matching nor conflicting training patterns are allowed to lie. Using these thresholds as the only user–adjustable parameters, the algorithm constructs the network dynamically and adjusts the radii individually. In short the main properties of the DDA algorithm are:

- **constructive training**: new neurons are added whenever necessary. The network is built from scratch: the number of required hidden units is determined during training; the individual radii of the Gaussians are adjusted dynamically during training.
- **fast training**: usually less than five epochs are needed to complete training due to the constructive nature of the algorithm.
- **guaranteed convergence**: the algorithm can be proven to terminate when a finite number of training examples is used [2].
- **two uncritical manual parameters**: only two parameters are required to be adjusted manually, fortunately the values of these two thresholds are not critical, as will be demonstrated in section 4.
- **distinct classification zones**: it can be shown that after training terminated, the network holds several conditions for all training patterns:
  - *class inclusion*: correct classifications are above a threshold θ^+, the correct–classification probability.
  - *class exclusion*: wrong classifications are below another threshold θ^- (misclassification probability).
  - *uncertainty*: patterns only residing in areas of conflict have low class–probabilities, providing an additional "don't know" answer.

These features make the application of PNNs to real world problems easy, since neither network architecture (i.e. number of hidden units) nor critical parameters have to be determined manually. Also the network's size does not grow linearly with the size of the training data as is the case for the classical PNN. This makes it possible to use redundant as well as large datasets for training. In addition the last feature enables the user to judge the confidence of the network. Providing an additional "don't know"–probability also enhances

the applicability of these networks in security sensitive scenarios where the usual "black box" nature of neural networks often permits their use.

3.1 The Algorithm

Operation of the DDA algorithm requires two distinct phases — training and classification. During the training phase, misclassified patterns either prompt the spontaneous creation of new RBF units (commitment) or the adjustment of conflicting RBF radii (shrinking of RBFs belonging to incorrect classes). To commit a new prototype, none of the existing RBFs of the correct class has an activation above θ^+ and, after shrinking, no RBF of a conflicting class is allowed to have an activation above θ^-.

Figure 3 (i–iv) shows an example that illustrates the first few training steps of the DDA algorithm: (i) a pattern of class A is encountered and a new RBF is created; (ii) a training pattern of class B leads to a new prototype for class B and shrinks the radius of the existing RBF of class A; (iii) another pattern of class B is classified correctly and shrinks again the prototype of class A; and (iv) a new pattern of class A introduces another prototype of that class.

After training is finished, two conditions hold for all input–output pairs $(\vec{x}, k)$ of the training data:

– at least one prototype of the correct class k has an activation value greater

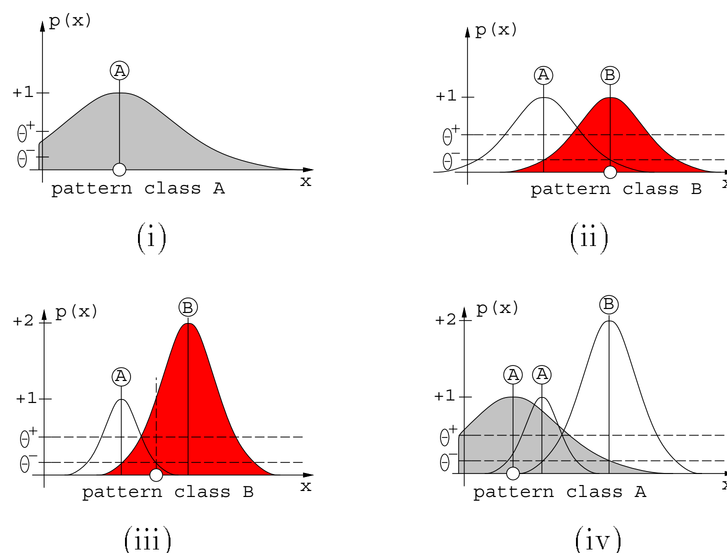

Fig. 3. An example of the DDA algorithm

```
        // reset weights:
(1)     FORALL prototypes p_i^k DO
           A_i^k = 0.0
        ENDFOR
        // train one complete epoch
(2)     FORALL training pattern (x⃗, k) DO:
(3)        IF ∃p_i^k : p_i^k(x⃗) ≥ θ⁺ THEN
(4)           A_i^k += 1.0
           ELSE
(5)           // "commit": introduce new prototype
              m_k += 1
(6)           μ⃗_{m_k}^k = x⃗
(7)           A_{m_k}^k = 1.0
(8)           σ_{m_k}^k = min_{l≠k, 1≤j≤m_l} { sqrt( - ||μ⃗_j^l - μ⃗_{m_k}^k||² / ln θ⁻ ) }
           ENDIF
           // "shrink": adjust conflicting prototypes
(9)        FORALL l ≠ k, 1 ≤ j ≤ m_l DO
              σ_j^l = min { σ_j^l , sqrt( - ||x⃗ - μ⃗_j^l||² / ln θ⁻ ) }
        ENDFOR
```

Fig. 4. The DDA algorithm for one epoch

than or equal to θ^+:

$$\exists i : p_i^k(\vec{x}) \geq \theta^+ \tag{6}$$

– all prototypes of conflicting classes have activations less than θ^- (m_l indicates the number of prototypes belonging to class l):

$$\forall l \neq k, 1 \leq j \leq m_l : p_j^l(\vec{x}) \leq \theta^- \tag{7}$$

The code to perform training for one epoch is shown in Figure 4, where p_i^k indicates prototype i of class k, A_i^k the corresponding weight (which models a local a priori probability of class k), $\vec{\mu}_i^k$ the center vector and σ_i^k the individual standard deviation. The algorithm operates as follows:

· before training an epoch, all weights A_i^k must be set to zero to avoid accumulation of duplicate information about the training patterns (1);
· next all training patterns are presented to the network (2);
· if the new pattern is classified correctly (3),
  the weight of the biggest prototype is increased (4),
· otherwise a new prototype is introduced (5);
  having the new pattern as its center (6),
  a weight equal to 1 (7), and

its initial radius ($\sigma^k_{m_k}$) is set as large as possible without misclassifying an already existing prototype of conflicting class (8) (At the beginning the new prototype will cover the entire feature space, because no conflicts arise.)

- the last step shrinks all prototypes of conflicting classes if their activations are too high for this specific pattern (9).

After only a few epochs (for practical applications, approximately five) the network architecture settles (no new commitment or adjustment), clearly indicating the completion of the training phase. Because radii of previously committed neurons can only shrink and never grow, it is easy to prove termination of this algorithm for a finite training dataset [2]. Due to the iterative nature of the training it is often possible to finally delete a few superfluous neurons that have a weight equal to zero. These neurons were inserted during an early stage of the training process and were replaced by more optimal neurons that cover a larger area of the feature space. After training is complete, the normalized output weights π^k_j can be computed from the prototype weights A^k_j through:

$$\forall 1 \leq k \leq c, \forall 1 \leq i \leq m_k \colon \pi^k_i = \frac{A^k_i}{\sum_{j=1}^{m_k} A^k_j}$$

For all experiments conducted to date, the choice of $\theta^+ = 0.4$ and $\theta^- = 0.2$ led to satisfactory results. In theory, those parameters should be dependent on the nature of the underlying model (discussed in the next section) but in practice, especially if the feature space is only sparsely occupied, the values of the two thresholds are not critical.

3.2 Choice of Thresholds

The only manually adjustable parameters on which the DDA algorithm is dependent are the two thresholds θ^+ and θ^-. Although the choice of $\theta^+ = 0.4$ and $\theta^- = 0.2$ has provided excellent results in practice, the influence that these parameters wield over the resulting classification boundaries is sometimes of interest. It is unusual for the training data to cover the entire feature space, and therefore the resulting classification of areas between training patterns is not determined. In this section it is investigated how the choices of θ^+ and θ^- influence the position of the class boundaries, as well as the resulting network size.

It is obvious that for $\theta^+ >> \theta^-$ many more neurons will be introduced than for $\theta^+ \approx \theta^-$. This is due to the larger conflict free zone; that is, the area where no patterns are allowed during training. Figure 5 demonstrates this effect using a one–dimensional example with patterns of two different classes `A` and `B`. Note

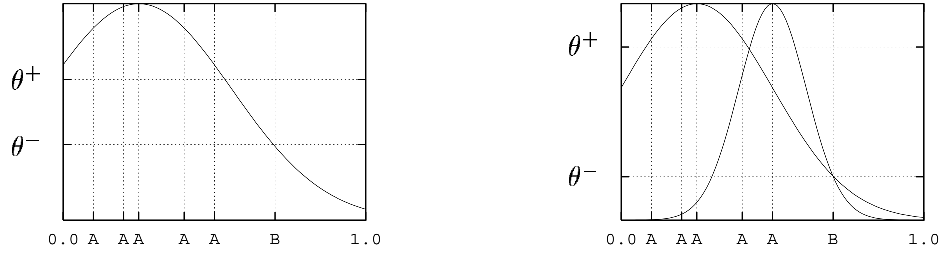

Fig. 5. Different choices for the thresholds lead to varying tolerance of the classification boundary and a changing number of required neurons.

how, with increasing distance between θ^+ and θ^-, the Gaussian covering the pattern at the border must move closer to the same and becomes less flexible in its positioning.

For θ^+ and θ^- being closer together the order of training examples can have a bigger influence on the position of the resulting class boundary. Figure 6 demonstrates this effect. The maximum distance a of the Gaussians' center to the training pattern at the border, depending on θ^+, θ^-, and the width of the pattern free zone b, computes to:

$$a = b\left(\frac{1}{\sqrt{\frac{\ln\theta^+}{\ln\theta^-}}+1} - 1\right)$$

This result can be used to compute the maximum distance c of the resulting class boundary from the middle of the pattern free area:

$$c = \frac{a}{a+2b}b = b\left(\frac{1}{\frac{1}{2}\sqrt{\frac{\ln\theta^+}{\ln\theta^-}}+1} - 1\right).$$

If the thresholds are chosen in a way to keep a small in comparison to b

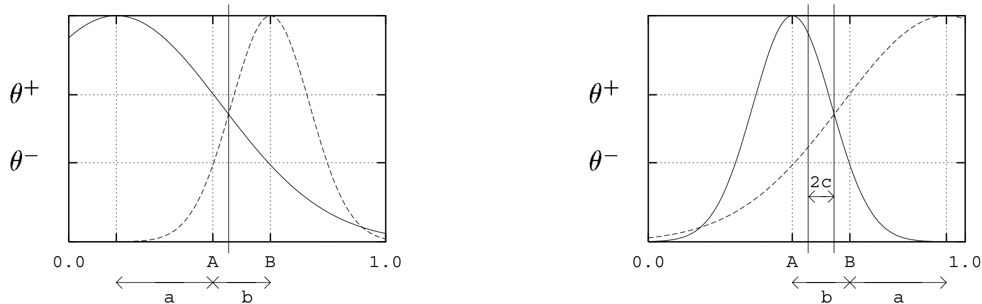

Fig. 6. Different orders of training examples can result in a variation of the resulting class boundaries.

the tolerance of the resulting class boundaries will be small. This means that the class boundaries will become more independent of the order of training examples when θ^+ is close to 1 and θ^- approaches 0.

In the previous paragraph it was assumed that class boundaries must be determined precisely. For real data sets these assumptions are too restrictive so that both parameters lose almost all their influence on the classification outcome. Therefore in practice the choice of both thresholds does not heavily influence the generalization capability of the resulting networks. The generalization error decreases slightly (together with a small increase in network size) when both thresholds are pulled apart, but the initial choice of $\theta^+ = 0.4$ and $\theta^- = 0.2$ leads to satisfactory results for most datasets. A small improvement can sometimes be achieved by observing the error on the training data and fine tuning both thresholds. For decision critical problems, a choice of $\theta^+ >> \theta^-$ will lead to networks with a finer generalization; that is the outputs for both classes will be close to 0 if the input is not close to one of the prototypes. In the next section a way to normalize the network's output that results in an additional indicator for this "don't know"–answer is presented.

3.3 Output Normalization

Most applications of Probabilistic Neural Networks for classification use some type of normalization that results in an output resembling a posteriori probabilities. This is usually achieved through:

$$p(\text{class } k|\vec{x}) = \frac{o_k(\vec{x})}{\sum_{l=1}^{c} o_l(\vec{x})} \tag{8}$$

This type of normalization is often motivated by the desire to achieve a partition of unity across the input space:

$$\forall \vec{x}\colon \sum_{k=1}^{c} p(\text{class } k|\vec{x}) = 1 \tag{9}$$

Unfortunately this type of normalization has a number of side effects [14]:

(i) *Loss of independence and change of shape.* The original shape of the basis functions is not only influenced by the choice of its weight but also by the proximity of other prototypes in the network.

(ii) *Coverage of the input space.* The whole input space is covered, not just the region that was defined using training data. This leads to a loss in locality, which is one of the intriguing features of PNNs.

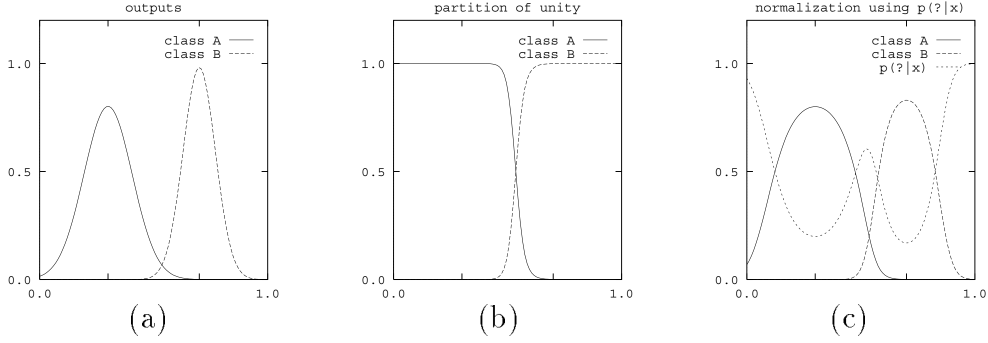

Fig. 7. Two different ways to normalize the network output.

(iii) *Reactivation and shift in maxima.* Using neighboring basis functions with different widths, the broader one will "reappear" on the other side of the thinner basis function. In addition, maxima of basis functions will not necessarily occur at the center of basis functions.

Of these issues, the second point seems to be especially ill suited for a network required to offer a measure of confidence along with its prediction. The method presented in this paper uses a different approach to compute normalization (see also [1]). In contrast to equation 9 an additional constant term $o_?$ is introduced:

$$\forall \vec{x} : \sum_{k=1}^{c} p(\text{class } k|\vec{x}) + o_? = 1 \tag{10}$$

Normalization in 8 therefore changes as follows, where $p(?|\vec{x})$ denotes the probability that the network cannot predict a class when $\vec{x}$ is observed:

$$p(\text{class } k|\vec{x}) = \frac{o_k(\vec{x})}{\sum_{l=1}^{c} o_l(\vec{x}) + o_?} \quad \text{and} \quad p(?|\vec{x}) = \frac{o_?}{\sum_{l=1}^{c} o_l(\vec{x}) + o_?} \tag{11}$$

In areas far from patterns that were observed during training, $o_?$ should be larger than all $o_k(\vec{x})$, resulting in a "don't know"–probability $p(?|\vec{x})$ close to one, and all class posteriori probabilities being almost zero. Figure 7 (c) shows this type of normalization in contrast to the common partition of unity (Figure 7 (b)) depending on the output of the network (Figure 7 (a)). Using the DDA for training, the selection of $o_?$ becomes straightforward: $o_? = \theta^-$. The reason for this choice is the semantic behind the two thresholds. Since θ^- defines the region of no conflict, outside of the θ^-–circle, $p(?|\vec{x})$ should be greater than $p(k|\vec{x})$. A normalization of this type enables the user to judge the confidence of the networks output, which is especially important for security sensitive applications.

4 Results

To demonstrate the behavior of the proposed DDA algorithm the well–known "two spiral" problem was used [5]. The required task involved discriminating between two intertwined spirals. For this experiment the spirals were changed slightly to make the problem more challenging. The original spirals' radii decline linearly and can be easily classified by PNNs with one global radius. To demonstrate the ability of the DDA algorithm to adjust the radii of each RBF individually, a quadratic decline was chosen for the radius of both spirals (see Figure 8). The training set consisted of 196 points, and the spirals made three complete revolutions:

$$r_i = \left(\frac{140 - i}{140}\right)^3 \ , \ x_i = r_i \sin\left(\frac{i * \pi}{16.0}\right) \ , \ y_i = r_i \cos\left(\frac{i * \pi}{16.0}\right) \ , \ 0 \leq i \leq 97.$$

After training of the classifier, $10,000$ equally distributed points in $[-1,1] \times [-1,1]$ were used for testing. Figure 9 shows the resulting classification of the feature space using a PNN trained through the DDA, a Multi Layer Perceptron (trained with RPROP [11], a fast version of Error Back Propagation) and a decision tree constructed through c4.5 [9]. Note that in all cases all training points are classified correctly. Three different ways to divide the feature space are easily observed. The DDA constructs an ensemble of round regions that approximate the curved class boundary well. Note that regions far away from the training points are grey, indicating a high "don't know"–probability. The network does not try to generalize in these areas. The Multi Layer Perceptron classifies the feature space using the typical hyper–planes that try to globally divide the feature space. On the left side of the picture it can be seen how the network classifies a stray towards another class. This effect was due to the particular order of the training examples and the used initialization but occurred in similar forms during other experiments as well. The decision tree algorithm c4.5 finally tries to classify through a series of axes–parallel decision lines that hierarchically divide the feature space. This approach is clearly not

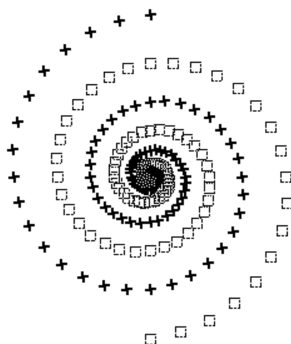

Fig. 8. The training data of the modified "two spiral" problem.

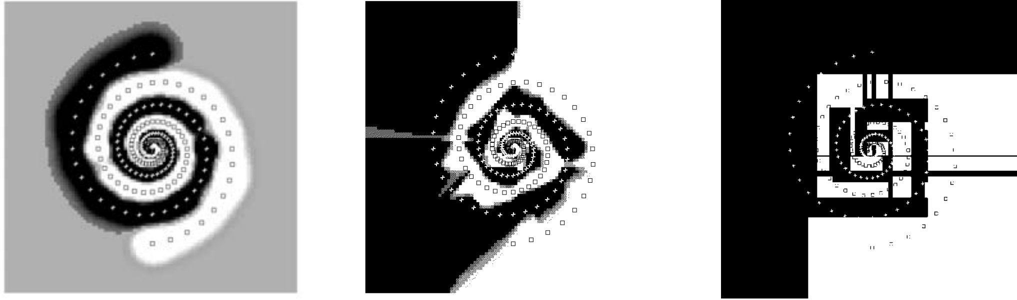

Fig. 9. The classification of the feature space using the DDA, RPROP, and c4.5.

suited to model the underlying structure of the two spirals.

This example does not necessarily indicate an overall superior performance of one classifier over others, but it gives a rough idea which types of problems will be better suited for one or the other classifier. The DDA will probably generalize better if class boundaries are intertwined, somewhat round, and local. Both the MLP and c4.5 will most likely model straight long boundaries better, or at least with a lower number of required resources (i.e. nodes in the network or depth of the decision tree).

To compare the DDA to other state–of–the–art classifiers on real–world datasets, eight benchmarks of the ESPRIT StatLog project [6] were used. In StatLog 20 different classifiers were applied to a variety of real–world datasets. Each algorithm was controlled by an expert, thus avoiding the usual bias towards favorites (for details see [6]). Eight of the publicly available datasets were chosen, and their characteristics are listed in Table 1. The first four smaller datasets were used through n–fold cross–validation; that is, each experiment was performed n times, using $(n-1)/n$–th of the dataset for training and the remaining $1/n$–th for testing. The reported error rates show the average over all n simulation runs. The larger datasets are specifically divided into a training set and a testing set.

In Table 2 the results of the k Nearest Neighbor, a Multi Layer Perceptron, the decision tree c4.5, the standard PNN, RCE together with its probabilistic

dataset	nr. attributes	nr. classes	number of patterns training	test
Diabetes	8	2	768	12–fold
Aust. Cred.	14	2	690	10–fold
Vehicle	18	4	846	9–fold
Segment	11	7	2310	10–fold
Shuttle	9	7	43,500	14,500
SatImage	36	6	4,435	2,000
DNA	240	3	2,000	1,186
Letter	16	26	15,000	5,000

Table 1. The used datasets of the StatLog–project.

dataset	DDA	StatLog results			other results		
		KNN	C4.5	MLP	PNN	RCE	P–RCE
Diabetes	**24.1**	32.4	27.0	24.8	24.9	39.8	25.8
Aust. Cred.	16.1	18.1	15.5	15.4	**13.5**	23.9	14.3
Vehicle	29.9	27.5	26.6	**20.7**	28.7	41.3	31.6
Segment	3.9	7.7	4.0	5.4	**3.5**	6.7	6.4
Shuttle	0.12	0.44	**0.10**	0.43	0.26	0.20	0.97
SatImage	**8.9**	9.4	15.0	13.9	9.8	15.2	10.7
DNA	16.4	14.6	**7.6**	8.8	10.5	46.0	11.5
Letter	6.4	6.8	13.2	32.7	**3.8**	12.9	5.8

Table 2. Error rates on the used StatLog datasets.

extension P–RCE, and the DDA are shown. The DDA algorithm shows very good performance on all but one dataset. DDA (as well as kNN) yields the poorest performance on the DNA data. This case consists of binary features, of which about 70% are not used for classification, and this proves to be a problem for Euclidean distance–based algorithms.

The classical PNN performs slightly better than the network generated by the DDA on about half of the datasets. It should be noted that the results for the classical PNN have been achieved after time–consuming fine tuning of the smoothing parameter σ and by using extensively large networks in some cases, since the number of neurons in the hidden layer is equivalent to the number of training examples. It can be easily demonstrated that the DDA algorithm produced significantly smaller networks than the classical PNN, particularly as the size of the dataset increases. Table 3 shows how the choice of the negative threshold ($\theta^+ = 0.4$ for all experiments) influences the size and performance of the resulting PNN for the Shuttle dataset. For large θ^-, neurons of different classes heavily overlap resulting in an error rate on the training data. This effect can be used to easily determine a good choice for θ^-. Less than 600 hidden units are sufficient to reach a classification performance comparable to the classical PNN with $45,000$ units. The classification error on the testdata of the DDA changes slightly by a factor of 4 when θ^- is varied over 5 orders of magnitude. Table 4 shows how the choice for the PNN's smoothing parameter σ influences the performance much more drastically.

number RBFs	number epochs	θ^-	Error (in %) training	test
396	4	0.1	0.67	0.65
685	3	0.01	0.15	0.21
865	3	1e-3	0.05	0.14
1016	3	1e-4	0.00	**0.12**
1058	3	1e-5	0.00	0.12
1091	3	1e-6	0.00	0.13

Table 3. The results of the DDA on the Shuttle dataset.

σ	Error (in %) training	test
0.5	21.59	20.84
0.1	7.69	7.54
0.05	2.09	2.01
0.01	0.44	0.52
0.005	0.09	0.26
0.001	0.00	0.61

Table 4. The results of the standard PNN on the Shuttle data.

All eight StatLog experiments were conducted with only few retrials to adjust θ^-. No validation set was used to determine the optimal setting; instead an observation of misclassified patterns in the training set was sufficient. Besides θ^-, no other manual adjustment was necessary; even termination of the training algorithm was controlled automatically.

5 Conclusions

A new algorithm to train Probabilistic Neural Networks has been proposed. In contrast to existing algorithms, large datasets can be used and the network's structure is determined automatically during training. In addition the utilization of individually adjusted radii for the hidden neurons reduces the size of the resulting network. There are only two, easy to adjust parameters that allow the user to distinguish between conflicting and matching prototypes at the training phase. The new algorithm trains very quickly, fewer than 6 epochs were sufficient to reach stability for all problems presented. Eight real world datasets from a large ESPRIT project were used to demonstrate the performance of the proposed DDA algorithm.

It is concluded that the DDA algorithm offers an easy to use methodology to quickly train Probabilistic Neural Networks offering state–of–the–art classification performance. Although the resulting networks are larger than comparable Multi Layer Perceptrons, they are significantly smaller than PNN topologies while still maintaining simple and speedy training with excellent classification results.

Acknowledgments

We thank the anonymous reviewers for their positive feedback. M. Berthold would like to thank Prof. D. Schmid for his support and the opportunity to work on this interesting project. Thanks also to Fay Sudweeks for the “australian touch”.

References

[1] Michael R. Berthold. A probabilistic extension for the DDA algorithm. In *International Conference on Neural Networks*, 1, pages 341–346. IEEE, 1996.

[2] Michael R. Berthold. *Konstruktives Training Probabilistischer Neuronaler Netze für die Musterklassifikation.* DISKI 155, infix Verlag, 1997.

[3] Michael R. Berthold and Jay Diamond. Boosting the performance of RBF networks with Dynamic Decay Adjustment. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems*, 7, pages 521–528, Cambridge MA, 1995. The MIT Press.

[4] Michael H. Hudak. RCE classifiers: Theory and practice. In *Cybernetics and Systems*, volume 23, pages 483–515. Hemisphere Publishing Corporation, 1992.

[5] K. Lang and M. Witbrock. Learning to tell two spirals apart. In *Proceedings of the Connectionist Summer School*, 1988.

[6] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, editors. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood Limited, 1994.

[7] John Moody and Christian J. Darken. Fast learning in networks of locally–tuned processing units. In *Neural Computation*, 1, pages 281–294. MIT, 1989.

[8] E. Parzen. On the estimation of a probability density function. In *Annals of Mathematical Statistics*, pages 1065–1076, 1962.

[9] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[10] Douglas L. Reilly, Leon N. Cooper, and Charles Elbaum. A neural model for category learning. In *Biological Cybernetics*, 45, pages 35–41, 1982.

[11] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *International Conference on Neural Networks*, 1, pages 586–591. IEEE, March 1993.

[12] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Exploration in the Microstructure of Cognition.* MIT Press, Cambridge, MA, 1987.

[13] S. Salzberg. A nearest hyperrectangle learning method. In *Machine Learning*, 6, pages 251–276, 1991.

[14] Robert Shorten and Roderick Murray-Smith. On normalizing radial basis functions networks. In *Proceedings of the fourth Irish Conference on Neural Networks, INNC'94*, pages 213–217, 1994.

[15] Patrick K. Simpson. Fuzzy min-max neural networks – part 1: Classification. *IEEE Transactions on Neural Networks*, 3(5):776–786, September 1992.

[16] Patrick K. Simpson. Fuzzy min-max neural networks – part 2: Clustering. *IEEE Transactions on Fuzzy Systems*, 1(1):32–45, January 1993.

[17] Donald F. Specht. Probabilistic neural networks. In *Neural Networks*, 3, pages 109–118, 1990.

[18] Donald F. Specht. Enhancements to probabilistic neural networks. In *International Joint Conference on Neural Networks*. IEEE, June 1992.

[19] Donald F. Specht. PNN: From fast training to fast running. In *Computational Intelligence, A Dynamic System Perspective*, pages 246–258. IEEE Press, 1995.

[20] Roy L. Streit and Tod E. Luginbuhl. Maximum likelihood training of probabilistic neural networks. *IEEE Transactions on Neural Networks*, 5(5):764–783, September 1994.

[21] Sebastian Thrun. Extracting rules from artificial neural networks with distributed representations. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems*, 7, pages 505–512, Cambridge MA, 1995. MIT Press.

[22] D. Wettschereck. A hybrid nearest-neighbour and nearest-hyperrectangle learning algorithm. In *Proceedings of the European Conference on Machine Learning*, pages 323–335, 1994.

Michael R. Berthold (M.Sc. 92, Ph.D. 97) was from 1993 to 1997 with the University of Karlsruhe and is currently a BISC Postdoctoral Fellow at the University of California, Berkeley. He was a Visiting Researcher at Carnegie Mellon University in 1991/92 and at Sydney University in 1994. He was also working as a Research Engineer at Intel Corp., Santa Clara in 1993. His current research interests include Neural Networks, Fuzzy Logic, and Intelligent Data Analysis.

Jay Diamond received his M.Sc. from the University of Manitoba in 1990 in the field of VLSI implementations of neural networks and joined Intel later that year to continue this work. He has worked in many roles at Intel from circuit and logic design to mobile architecture to technical marketing. He is currently the Technical Strategist for Intel's New Media Programs group, which produces Mediadome(sm), a website which merges name–brand media with cutting–edge technology to deliver regularly scheduled interactive internet programming.