Output Partitioning of Neural Networks

Sheng-Uei Guan[*], Qi Yinan, Syn Kiat Tan and Shanchun Li

Department of Electrical and Computer Engineering, National University of Singapore

10 Kent Ridge Crescent, Singapore 119260

Abstract Many constructive learning algorithms have been proposed to find an appropriate network structure for a classification problem automatically. Constructive learning algorithms have drawbacks especially when used for complex tasks and modular approaches have been devised to solve these drawbacks. At the same time, parallel training for neural networks with fixed configurations has also been proposed to accelerate the training process. A new approach that combines advantages of constructive learning and parallelism, *output partitioning*, is presented in this paper. Classification error is used to guide the proposed incremental-partitioning algorithm, which divides the original dataset into several smaller sub-datasets with distinct classes. Each sub-dataset is then handled in parallel, by a smaller constructively trained sub-network which uses the whole input vector and produces a portion of the final output vector where each class is represented by one unit. Three classification datasets are used to test the validity of this method, and results show that this method reduces the classification test error.

Index Terms — constructive learning algorithm, neural networks, output partitioning

[*] Corresponding author, e-mail address: eleguans@nus.edu.sg. The authors are with the Department of Electrical and Computer Engineering, National University of Singapore, 10 Kent Ridge Crescent, Singapore 119260.

1. Introduction

It is widely known that network size is of crucial importance for neural networks. Too small a network cannot learn the problem well [1], while a size too large will lead to overfitting and thus poor generalization [2]. It is a key issue in neural network design to find an appropriate network size automatically for a given application and optimize the set of network weights.

There are three approaches to tackle this issue: *pruning*, *regularization*, and *constructive algorithms*. In pruning [3], some hidden units or weights are removed during training if they are no longer actively used. Regularization uses some penalty terms in the cost function to force the weights to yield smooth approximations [4]. The third approach, called *growing* or *constructive approach*, starts with a small network and then grows additional hidden units and weights until a satisfactory solution is found. The constructive approach has a number of advantages over pruning and regularization approaches. Detailed descriptions can be found in [5].

Constructive methods include the *Dynamic Node Creation* (DNC) method [6], CasCor family of learning algorithms [7] (including standard *Cascade-Correlation* (CC) algorithm [8]), *Constructive single-hidden-layer network* [9] and *Constructive Backpropagation* (CBP) algorithm [10], etc. Among them, DNC and CBP have only one single hidden layer and a new hidden unit receives complete connections from the inputs and is connected to all output units. DNC has no shortcut connections between the input units and the output units while CBP begins with shortcut connections. CC begins with shortcut connections and then automatically trains and adds new hidden units one by one to create a multilayered network. Each hidden layer so constructed consists of just one single unit, which receives connections from each of the network's inputs as well as from all hidden units in previous layers.

Efforts [7, 9-13] have been made to compare their generalization capability. The results obtained show that: 1) In most cases, whether to cascade hidden units or not does not make a significant difference at all; 2) For some datasets, especially for regression problems, non-cascading hidden units is even superior to cascading them.

Besides constructive methods above, Bayesian approaches [27] can be appropriate alternatives to automatically determine the optimal network size. In [28, 29], Mackay suggested a hierarchical inference approach with the following three progressive levels: weight inference, hyperparameter inference and model comparison. Using Bayesian approaches, model comparisons do not require any validation sets, hence more samples are available for training. However, Bayesian approaches are in general computationally expensive.

Although constructive learning algorithms can automatically find an optimal combination, they are still suffering from drawbacks such as inefficiency in utilizing network resources as the task (and the network) gets larger, and inability of the current learning schemes to cope with high-complexity tasks [14]. Large networks tend to introduce high internal interference because of the strong coupling among their input-to-hidden layer weights [15]. Modular neural networks attempts to solve these issues via a "divide and conquer" approach. Using this approach, [16] divides the training set into subsets recursively using hyperplanes until all the subsets become linearly separable. [17]

constructs neural networks where the first unit introduced on each hidden layer is trained on all examples and further units on the layer are trained primarily on examples not already correctly classified. Using modular neural networks, the input data are partitioned into several subspaces, and simple systems are trained to fit the local data. Such data partitioning is often more effective than training on the whole input data space [18].

At the same time, *parallel training* has also been proposed in order to gain faster training. For multilayered neural networks, backpropagation algorithms, including BP [19], RPROP [20], Quickprop [21], SuperSAB [22], etc., reveal four different types of parallelism as follows [23]: *training session parallelism*, *training set parallelism*, *pipelining* and *node parallelism*. These four parallelisms are commonly used for parallel training of a neural network with fixed configuration.

Different from the previously proposed modular neural networks and the four network parallelisms mentioned above we propose a new approach — *output partitioning*. A dataset to be classified can be partitioned into several smaller sub-datasets with distinct classes. Each sub-dataset is then handled by a smaller sub-network which uses the whole input vector as input and produces a portion of the final output vector (each class is represented by a unit). Each sub-network solution to each sub-dataset is grown and trained using constructive algorithms; this can be performed simultaneously on parallel processing elements. The grown sub-networks are then integrated to produce the final results. This method creates smaller neural networks which have reduced internal interference among hidden layers, consequently, reduces computational time and improves performance. In section 1, we briefly recall the CBP learning algorithm. The concept of output partitioning is described in section 2. The proposed incremental-partitioning algorithm is then described in section 3. Experiments about output partitioning are implemented and the results analyzed in section 4. Finally, the conclusions are presented in section 5.

1.1 Constructive backpropagation learning algorithm

The CBP learning algorithm (depicted in Figure 1) can be described briefly as follows [10]:

1. *Initialization*: The network has no hidden units initially. Only bias weights and shortcut connections from input units to output units feed the output units. The weights of this initial configuration are trained by minimizing the *sum of squared errors* cost function:

$$SSE = \sum_{p=1}^{P}\sum_{k=1}^{K}(o_{pk} - t_{pk})^2 \qquad (1)$$

where P is the number of training patterns, K is the number of output units, o_{pk} is the actual output value at the kth output node for the pth training pattern and t_{pk} is the desired output value at the kth output node for the pth training pattern. After training, the weights of this initial configuration are fixed permanently.

2. *Training a new hidden unit*: Connect inputs to the new unit (let the new unit be the ith hidden unit, $i > 0$) and connect its output to the output units. Adjust all the weights connected to the new unit (both input/output connections) by minimizing modified SSE (mSSE):

$$mSSE_i = \sum_{p=1}^{P}\sum_{k=1}^{K}\left(a(\sum_{j=0}^{i-1} w_{jk} o_{pj} + w_{ik} o_{pi}) - t_{pk} \right)^2 \qquad (2)$$

where w_{jk} is the connection from the j th hidden neuron to the k th output unit (w_{0k} represents a set of weights which are the bias weights and shortcut connections trained in step 1), o_{pj} is the output of the j th hidden neuron for the p th training pattern (o_{p0} represents inputs to bias weights and shortcut connections), while $a(\cdot)$ is the activation function. Note that from the new i th neuron's perspective, the previous neurons are fixed. We are only training the weights connected to the new unit.

3. *Freezing the new hidden unit*: Fix the weights connected to the units permanently.

4. *Testing for convergence*: If the current number of hidden units yields an acceptable solution, then stop training. Otherwise go back to step 2.

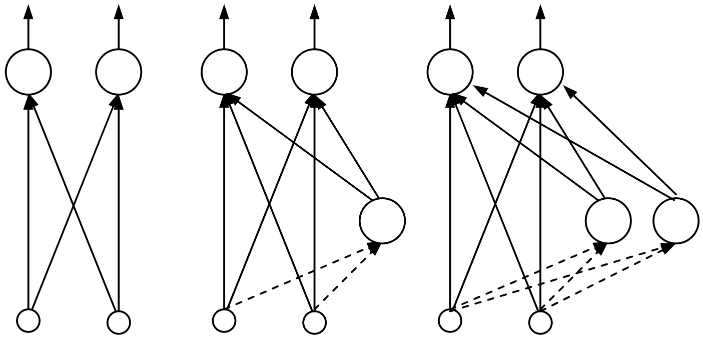

Figure 1. Evolution of the CBP architecture

2. Output partitioning

The goal for constructively constructing a neural network with output partitioning is to obtain an appropriate architecture of sub-networks with a set of weights that satisfy $E < E_{th}$ (E_{th} is the threshold of E).

2.1 Dataset partitioning

If we partition the output vector (each class is represented by one unit) into r sections ($S_1, S_2, \cdots, S_r$), each containing at least one output unit, then equation (1) can be transformed into:

$$
\begin{aligned}
E &= \sum_{p=1}^{P}\sum_{k=1}^{K}(o_{pk} - t_{pk})^2 \\
&= \sum_{p=1}^{P}\left[\sum_{k_1=1}^{S_1}(o_{pk_1} - t_{pk_1})^2 + \sum_{k_2=S_1+1}^{S_1+S_2}(o_{pk_2} - t_{pk_2})^2 + \cdots + \sum_{k_r=S_1+S_2+\cdots+S_{r-1}+1}^{K}(o_{pk_r} - t_{pk_r})^2\right] \\
&= \sum_{p=1}^{P}\sum_{k_1=1}^{S_1}(o_{pk_1} - t_{pk_1})^2 + \sum_{p=1}^{P}\sum_{k_2=S_1+1}^{S_1+S_2}(o_{pk_2} - t_{pk_2})^2 + \cdots + \sum_{p=1}^{P}\sum_{k_r=S_1+S_2+\cdots+S_{r-1}+1}^{K}(o_{pk_r} - t_{pk_r})^2 \\
&= E_1 + E_2 + \cdots + E_r \qquad (3)
\end{aligned}
$$

where $S_1 + S_2 + \cdots + S_r = K$.

E_1, E_2 ,…, E_r are independent of each other. The only constraint among them is that their sum E should be smaller than E_{th} . So we can partition the original dataset into r sub-datasets.

2.2 Sub-network growing

After partitioning, the original dataset is divided into r sub-datasets. The original single neural network solution for the dataset is replaced by r sub-networks (sub-NN), i.e. sub-NN1, sub-NN2, …, sub-NNr, each of which is constructed for a sub-dataset, as shown in Figure 2. Each sub-NN can be grown and trained on different processing elements. When we are classifying an unknown sample, each sub-NN computes a portion of the output vector and their results are merged to generate the final output.

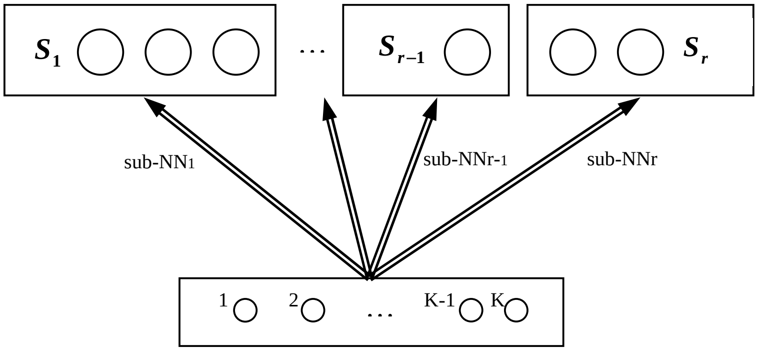

Figure 2. Parallel growing based on output partitioning

3. Algorithm for growing and training neural networks using output partitioning

3.1 Definitions

Constructive learning algorithms are very sensitive to changes in the stopping criteria. If training is too short, the components of the network will not work together well enough for good results. If training is too long, it costs too much computation time and may result in overfitting and bad generalization. In this paper, we adopt the method of *early stopping* [7, 24] using a validation set.

The set of available patterns is divided into three sets: a *training set* is used to train the network, a *validation set* is used to evaluate the quality of the network during training and to measure overfitting, and finally a *test set* is used at the end of training to evaluate the resultant network. In this paper, the size of the training, validation and test set is 50%, 25% and 25% of the dataset's total available patterns.

The error measure E used is *the squared error percentage* [24], derived from the normalization of the mean squared error to reduce the dependence on the number of coefficients in (1) and on the range of output values used:

$$E = 100 \cdot \frac{o_{\max} - o_{\min}}{K \cdot P} \sum_{p=1}^{P} \sum_{k=1}^{K} (o_{pk} - t_{pk})^2 \qquad (4)$$

where $o_{\max}$ and $o_{\min}$ are the maximum and minimum output values in (1).

$E_{tr}(t)$ is the average error per pattern of the network over the training set, measured after epoch t. The value $E_{va}(t)$ is the corresponding error on the validation set after epoch t and is used by the stopping criteria. $E_{te}(t)$ is the corresponding error on the test set; it is not known to the training algorithm but characterizes the quality of the network resulting from training. The value $E_{opt}(t)$ is defined to be the lowest validation set error obtained in epochs up to epoch t:

$$E_{opt}(t) = \min_{t' \le t} E_{va}(t') \qquad (5)$$

The relative increase of the validation error over the minimum so far (in percent) is defined as the *generalization loss* at epoch t:

$$GL(t) = 100 \cdot (\frac{E_{va}(t)}{E_{opt}(t)} - 1) \qquad (6)$$

A high generalization loss is one candidate reason to stop training because it directly indicates overfitting.

A *training strip of length* m [24] is defined to be a sequence of m epochs numbered n+1, …, n+m, where *n* is divisible by m. The training *progress* measured after a training strip is:

$$P_m(t) = 1000 \cdot \left(\frac{\sum_{t' \in t-m+1 \ldots t} E_{tr}(t')}{m \cdot \min_{t' \in t-m+1 \ldots t} E_{tr}(t')} - 1\right) \qquad (7)$$

that is used to measure how much larger the average training error is than the minimum training error during the training strip.

3.2 Procedure for growing and training of the sub-networks

The procedure for growing and training each sub-NN is shown in Figure 3. s*ub_epoch* is used to represent running epochs for training one configuration of a sub-NN. *total_epoch* is used to represent the total running epochs for growing the sub-NN (the sum of all *sub_epochs*). Initially, the sub-NN has no hidden units. There are only bias weights and the shortcut connections between inputs and output units. Now *sub_epoch* and *total_epoch* are both set to 1. Then train the initial neural network using the RPROP algorithm. Set the corresponding validation error $E_{va} = E_{opt}$, the optimal validation error so far. Record the weights accordingly as the optimal weights.

After every *m* epochs (*m* is strip length, we used *m* =5), compare E_{va} to E_{opt}. If E_{va} is less than E_{opt}, then set E_{opt} as E_{va} and record the weights accordingly as the optimal weights. After every epoch, check the overall stopping criteria :

(Training has reached Z = 5000 epochs) ***OR*** *(* $E_{opt} < E_{th}$ *)*

OR *((Reduction of training set error due to the last new hidden unit is less than 1%)*

AND *(Validation set error increased due to the last new hidden unit))*

If the overall stopping criteria are not satisfied, then check the criteria for adding a new hidden unit. If a new hidden unit should be added, then copy weights from the optimal weights and freeze the weights. The *criteria for adding a new hidden unit*:

(At least X epochs reached)

AND ((Generalization loss $GL(t)$ >5) **OR** (Training progress $P_m(t)$ <0.1) **OR** (Y epochs reached))

```
Construct and Initialize sub-NN
sub_epoch = 1
total_epoch = 1
While  (total_epoch < Z)
{
        Train the current configuration of sub-NN for one epoch
        If (sub_epoch == 1)
                $E_{opt} = E_{va}$  (Record the weights accordingly as the optimal weights)
        If (sub_epoch % $m$ == 0 && $E_{va} < E_{opt}$)
                $E_{opt} = E_{va}$  (Record the weights accordingly as the optimal weights)
        If ( $E_{opt} < E_{th}$ || little-improvement from last new hidden unit)
                Break
        If (sub_epoch > X && ( $GL(t)$ >5 || $P_k(t)$ <0.1 || sub_epoch > Y))
                Copy weights from the optimal weights
                Add a new hidden unit and initialize the weights (randomly)
                sub_epoch = 0
        sub_epoch ++
        total_epoch++
}
Calculate $E_{te}$ and Exit
#We used X = 80 epochs while Y = 500 epochs.
```

Figure 3. The growing and training procedure for sub-NNs

3.3 Incremental-Partitioning algorithm

The motivation to use several smaller sub-NNs is to reduce the high internal interference inherent in large networks [15] which has to classify all K classes in a particular dataset. By dividing the original dataset into several sub-datasets with distinct classes and using smaller separate, sub-NNs for each sub-dataset of classes, we can reduce the internal interferences in the input-to-hidden layer weights, hence reduce the overall classification errors. Classification error is used as a metric to measure the interference among different classes. Here we propose an incremental-partitioning algorithm to produce near-optimal partitioning of classes according to the classification error. The algorithm produces a near optimal partitioning of the classes, in the form of an unordered list – for example {{6,3,2,11,9},{10,1,8},{4,7},{5}} where each element in the list represents a specific class in the original dataset (11 classes). In the above example, there are four partitions each representing a sub-dataset that will be trained on a sub-NN separately.

The details of this algorithm are described as follows.

Step 1: Find the classification error C_i of each class and order them in ascending order as $\{C_a,...,C_b,...,C_c\}$, where $C_a \le C_b \le C_c$. To obtain the individual C_i, $1 \le i \le K$, all patterns not belong to class i are labeled as patterns of class $\bar{i}$. A single NN is then used for the resulting two-class classification problem.

Step 2: Form the first partition with the class i, i.e. $\{\{i\}\}$.

Step 3: Choose another class j, add it to the above partition, i.e. $\{\{i, j\}\}$ and measure the classification error. Similar to Step 1, all patterns not belonging to class i and class j are assigned as patterns of class $\overline{i, j}$. A single NN is then used for the resulting three-class classification problem. Next, put class j into a new partition, i.e. $\{\{i\},\{j\}\}$ and measure again the classification error (training a sub-NN for each partition). For each sub-NN training respectively, all patterns not belonging to class i/class j are assigned as patterns of class $\bar{i}$/class $\bar{j}$. A single NN is then used for each resulting two-class classification problem. Adopt the partitioning that produces a smaller classification error.

Step 4: Repeat Step 3 for all remaining classes in the dataset. Assign another class to each existing partition and measure the resulting classification error. Next, form a new partition containing only this class and again measure the resulting classification error. For each sub-NN to be trained, all patterns not belonging to the classes in the partition will be assigned as patterns of a single "dummy" class. The classification error thus obtained is a measure of the difficulty in discriminating between patterns of a particular class (or some classes) and patterns from all the other classes. Based on the smallest possible classification error over all the possible combinations tested, this class is then either added to any existing partition or used to form a new single-class partition. In the final list, each partition contains distinct classes to be classified using a separate sub-NN.

There are two major considerations required in the above algorithm - how to choose the first class to form the first partition and how to decide the order of assigning the remaining classes.

For the first class, we chose the class with the largest classification error (last element in $\{C_a, ..., C_b, ..., C_c\}$) since this class will dominate both the NN performance and the NN structure. When the classification error for a particular class is high, it will be very difficult to distinguish between the patterns belonging to this class and those patterns from the other classes (see Step 1 on how C_i is obtained). Therefore, the classification error for patterns from this class will dominate the overall classification error. The size of the classification error also influences directly the overall structure of a NN. During NN training, the NN either generates more hidden units or performs more weight adjustment to reduce any large classification error.

We assign the remaining classes using the ordered list $\{C_a, ..., C_b, ...\} \setminus C_c$ (the class with largest classification error has previously been assigned), that is to try and partition the classes with the smallest classification error first. Patterns belonging to classes with smaller errors tend to be more easily distinguished from patterns belonging to the other classes. And it is more likely to obtain the lowest possible classification error when such classes with smaller errors are assigned into existing partitions. Furthermore, lesser partitions will be present in the final list generated. With lesser sub-NNs to be trained, fewer resources are required.

Soft constraints were also used in the incremental-partitioning process where a small predefined performance margin is used for considering alternative solutions. During each stage of assigning classes into partitions, there can exist several intermediate solutions whose classification errors are close. Whenever a big increase in the classification error for all possible solutions is encountered in any stage of the algorithm, back-tracking is performed and we consider alternative intermediate solutions (with classification errors within the margin) obtained in the previous stage.

4. Experimental results

4.1 Experiment scheme

In this paper, we adopt the CBP network growing algorithm with the RPROP algorithm [20] for training the sub-NNs. The RPROP algorithm was used with the following parameters: $\eta^{+} = 1.2$, $\eta^{-} = 0.5$, $\Delta_0 = 0.1$, $\Delta_{\max} = 50$, $\Delta_{\min} = 1.0e - 6$, initial weights from –0.25 … 0.25 randomly. The hidden units and output units all use the sigmoid activation function. All the experiments were simulated on a single Pentium III –500 PC. If implemented in parallel on multiple processors, communication overhead may be negligibly small as there is not much communication until the result merging stage.

4.2 Benchmark dataset descriptions

We examined the above automatic partitioning procedure on three classification datasets (see Table 1 for description). For each classification dataset, we use the incremental-partitioning algorithm with performance margin 1% (i.e. alternative solutions have classification errors within 1% of the lowest classification error), as given in Section 3.2.

Table 1. Benchmark dataset descriptions

Dataset	No. of outputs (classes)	No. of inputs	No. of examples in (train / validation / test) sets
Thyroid	3	20	3600 / 1800 / 1800
Glass	6	9	107 / 54 / 53
Vowel	11	10	495 / 248 / 247

4.2.1 Thyroid1

The individual classification error of each class is shown in Table 2. According to the algorithm in Section 3.2, we form the first partition with class 2. The incremental-partitioning process is shown in Table 3.

Table 2. Classification errors for individual classes in Thyroid1

Class	1	2	3
C_i	1.58	**1.83**	1.72

Table 3. Incremental-partitioning of Thyroid1

Class Assigned	Partitioning / C_i	Partitioning / C_i	Partitioning / C_i
2	{2} / 1.83		
1	{2,1} / 2.22	**{{2},{1}} / 1.89**	
3	{{2,3},{1}} / 2.06	**{{2},{1,3}} / 1.72**	{{2},{1},{3}} / 1.89

The solutions formed during each stage, together with their associated classification error C_i, are shown in Table 3. The class being assigned is shown in the leftmost column and the classification errors of partitionings attempted by the algorithm are recorded in the subsequent columns with the best current partitioning highlighted in bold. As the algorithm progresses, more partitionings will be attempted if more partitions were formed in the earlier stages. The partitionings can be read as, for example {2,1} means class 2 and 1 are assigned into one same partition and {{2},{1}} means class 2 and 1 are assigned into two different partitions. For the Thyroid1 problem, we get the near-optimal partition of {{2},{1, 3}}.Compared with the non-partitioning and full-partitioning methods (shown in Table 4), the classification error was reduced by 8.9% and 7.5% respectively.

Table 4. Comparison of different methods for Thyroid1

	Non-partitioning	Full-partitioning	Incremental-partitioning
Classification error	1.86	1.89	1.72

4.2.2 Glass1

The individual classification error of each class is shown in Table 5. We form the first partition with class 2. The incremental-partitioning process is shown in Table 6.

Table 5. Classification errors for individual classes in Glass1

Class	1	2	3	4	5	6
C_i	20.28	**34.91**	8.30	1.04	0.85	9.43

Table 6. Incremental-partitioning of Glass1

Class Assigned	Partitioning / C_i	Partitioning / C_i	Partitioning / C_i	Partitioning / C_i
2	**{2} / 34.91**			
5	**{2,5} / 34.34**	**{{2},{5}} / 34.15**		
4	{2,5,4} / 37.74	{{2,5},{4}} / 36.91	**{{2,4},{5}} / 35.00**	{{2},{4,5}} / 35.85
	{{2},{4},{5}} / 34.82			
3	{{2,3},{4},{5}} / 30.76	{{2,}{3,4},{5}} / 34.49	{{2},{4},{3,5}} / 35.57	**{{2},{3},{4},{5}} / 29.06**
	{{2,3,4},{5}} / 32.08e	{{2,4},{3,5}} / 41.51	{{2,4},{3},{5}} / 43.39	
6	**{{2,6},{3},{4},{5}} / 36.42**	{{2},{3,6},{4},{5}} / 37.08	{{2},{3},{4,6},{5}} / 37.08	{{2},{3},{4},{5,6}} / 39.05
	{{2},{3},{4},{5},{6}} / 36.13			
1	**{{2,6,1},{3},{4},{5}} / 32.93**	{{2,6},{3,1},{4},{5}} / 36.96	{{2,6},{3},{4,1},{5}} / 37.73	{{2,6},{3},{4},{5,1}} / 38.85
	{{2,1},{3},{4},{5},{6}} / 36.32	{{2},{3,1},{4},{5},{6}} / 36.96	{{2},{3},{4,1},{5},{6}} / 39.84	{{2},{3},{4},{5,1},{6}} / 37.71
	{{2,},{3},{4},{5},{1,6}} / 39.88			

As shown in Table 6, in some stages, partitionings whose classification errors are within the specified performance margin (1%), are also kept for future generation (more than one entry highlighted in bold). Compared with the non-partitioning and full-partitioning methods (shown in Table 7), the classification error was reduced by 8.9% and 20% respectively.

Table 7. Comparison of different methods for Glass1

	Non-partitioning	Full-partitioning	Incremental-partitioning
Classification error	41.22	36.13	32.93

4.2.3 Vowel1

The individual classification error of each class is shown in Table 8. We form the first partition with class 6. The incremental-partitioning process is shown in Table 9.

Table 8. Classification errors for individual classes in Vowel1

Class	1	2	3	4	5	6	7	8	9	10	11
C.error	2.00	2.39	2.00	2.81	5.43	**8.40**	4.72	3.95	7.45	2.19	6.84

Table 9. Incremental-partitioning of Vowel1

Class Assigned	Partitioning / C_i	Partitioning / C_i	Partitioning / C_i
6	**{6} / 8.40**		
3	**{6,3} / 8.40**	{{6},{3}} / 11.84	
1	**{{6,3},{1}} / 13.26**	{6,3,1} / 15.69	
10	{{6,3,10},{1}} / 15.49	{{6,3},{10},{1}} / 13.82	**{{6, 3},{10},{1}} / 12.85**
2	**{{6,3,2},{10,1}} / 13.82**	{{6,3},{10,1,2}} / 14.63	{{6,3},{10,1},{2}} / 14.37
4	{{6,3,2,4},{10,1}} / 14.47	{{6,3,2},{10,1,4}} / 13.77	**{{6,3,2},{10,1},{4}} / 13.21**
8	{{6,3,2,8},{10,1},{4} / 19	**{{6,3,2},{10,1,8},{4}} / 13.92**	{{6,3,2},{10,1},{4,8}} / 17.71
	{{6,3,2},{10,1},{4},{8}} / 17.41		
7	{{6,3,2,7},{10,1,8},{4}} / 22.98	{{6,3,2},{10,1,8,7},{4}} / 20.39	**{{6,3,2},{10,1,8},{4,7}} / 19.08**
	{{6,3,2},{10,1,8},{4},{7}} / 20.60		
5	{{6,3,2,5},{10,1,8},{4,7}} / 18.72	{{6,3,2},{10,1,8,5},{4,7}} / 16.85	{{6,3,2},{10,1,8},{4,7,5}} / 19.64
	{{6,3,2},{10,1,8},{4,7},{5}} / 16.09		
11	**{{6,3,2,11},{10,1,8},{4,7},{5}} / 21.15**	{{6,3,2},{10,1,8,11},{4,7},{5}} / 25.40	{{6,3,2},{10,1,8},{4,7,11},{5}} / 27.53
	{{6,3,2},{10,1,8},{4,7},{5,11}} / 21.41	{{6,3,2},{10,1,8},{4,7},{5},{11}} / 22.67	
9	**{{6,3,2,11,9},{10,1,8},{4,7},{5}} / 18.57**	{{6,3,2,11},{10,1,8,9},{4,7},{5}} / 28.34	{{6,3,2,11},{10,1,8},{4,7,9},{5}} / 20.11
	{{6,3,2,11},{10,1,8},{4,7},{5,9}} / 19.43	{{6,3,2,11},{10,1,8},{4,7},{5},{9}} / 21.15	

We obtain the near-optimal partitioning {{6, 3, 2, 11, 9}, {10, 1 8}, {4, 7}, {5}}. The comparison with the non-partitioning and full-partitioning methods is shown in Table 10. We can find the classification error was reduced significantly by partitioning. Recalling from Table 9, an interesting result is that we can find many solutions whose error is smaller than the non-partitioning or full-partitioning methods. For this dataset, there exist several incremental-partitioning solutions.

Table 10. Comparison of different methods for Vowel1

	Non-partitioning	Full-partitioning	Incremental-partitioning
Classification error	34.73	24.39	18.57

4.3 Analysis of complexity

In the proposed algorithm, the computation complexity is not constant but changes with the specified performance margin. When soft constraints are not applied, the required number of partitions to test is $\sum_{i=1}^{K} i$, where K is the total number of classes. If the specified performance margin is large, the number of competing partitioning candidates per stage will increase. In the extreme situation where the margin is infinite, the searching algorithm will search every possible partitioning to look for the global optima. Therefore, we can find a partitioning with the best performance. If the margin is small, for example 1% in above cases, the number of competing partitions is decreased. So the complexity of computation is also decreased. However, the final partitioning is less probable to be the global optima. In conclusion, there is a tradeoff between the complexity of computation and the performance of the incremental-partitioning algorithm.

5. Conclusions

An approach to grow and train neural networks based on output partitioning is presented**.** A dataset can be partitioned into several simpler sub-datasets where internal interference is greatly reduced. From the experimental results obtained for all three datasets, partitioning a dataset using our proposed incremental-partitioning algorithm leads to lower classification errors. The results were better than either the non-partitioning and full-partitioning methods. The improvement is especially significant in datasets with more classes (Vowel). In datasets with large number of classes, the chances of having imbalanced class data are higher. The actual distance between each class is also likely to be non-uniform. Output partitioning utilizes the above phenomenon to assign appropriate classes to separate sub-NNs with the incremental-partitioning algorithm. And through adjusting the performance margin, we have tradeoff between the computation complexity and the performance of the final partitioning.

References

[1] A. Blum and R. L. Rivest, Training a 3-node neural network is NP-complete, *Neural Networks*, 5(1), 1992, pp. 117-128.

[2] E. B. Baum and D. Haussler, What size net gives valid generalization? *Neural Computation*, 1(1), 1989, pp. 151-160.

[3] R. Reed, Pruning algorithm—a survey, *IEEE Transactions on Neural Networks*, 4(5), 1993, pp. 740-747.

[4] T. Poggio, and F. Girosi, Regularization algorithms for leaning that are equivalent to multi-layer networks, *Science*, 247, 1990, pp. 978-982.

[5] T. Y. Kwok and D. Y. Yeung, Objective functions for training new hidden units in constructive neural networks, *IEEE Transactions on Neural Networks*, 8(5), 1997, pp. 1131-1148.

[6] T. Ash, Dynamic node creation in backpropagation networks, *Connection Science*, 1(4), 1989, pp. 365-375.

[7] L. Prechelt, Investigation of the CasCor family of learning algorithms, *Neural Networks*, 10(5), 1997, pp. 885-896.

[8] S. E. Fahlman and C. Lebiere, The cascade-correlation learning architecture, In D. S. Touretzky, *Advances in neural information processing systems 2*, Morgan Kaufmann Publishers, CA, 1990, pp. 524-532.

[9] D. Y. Yeung, A neural network approach to constructive induction, *Proceedings of the Eighth International Workshop on Machine Learning,* Evanston, Illinois, 1991, U.S.A.

[10] M. Lehtokangas, Modelling with constructive backpropagation, *Neural Networks,* 12, 1999, pp. 707-716.

[11] S. Sjogaard, Generalization in cascade-correlation networks, *Proceedings of the IEEE signal processing workshop*, 1992, pp. 59-68.

[12] J. Yang and V. Honavar, Experiments with cascade-correlation algorithm, Technical report *91-16*, Department of Computer Science, Iowa State University, 1991.

[13] C. S. Squires and J. W. Shavlik, Experimental analysis of aspects of the cascade-correlation learning architecture, *Machine learning research group working paper 91-1*, Computer Science Department, University of Wisconsin-Madison, 1991.

[14] G. Auda, M. Kamel and H. Raafat, Modular neural network architectures for classification, *IEEE International Conference on Neural Networks*, V2, 1996, pp. 1279-1284.

[15] R. A. Jacobs and M. I. Jordan *et al.*, Adaptive mixtures of local experts, *Neural Computation*, 3(1), 1991, pp. 79-87.

[16] P. Liang, Problem decomposition and subgoaling in artificial neural networks, *IEEE International Conference on Systems, Man and Cybernetics*, Los Angeles, CA. 1990, pp.178-181.

[17] S. G. Romaniuk and L. O. Hall, Divide and conquer neural networks, *Neural Networks*, V6, 1993, pp. 1105-1116.

[18] A. J. C. Sharkey, Modularity, combining and artificial neural nets, *Connection Science*, 9(1), 1997, pp. 3-10.

[19] D. E. Rumelhart, G. Hinton and R. Williams, Learning internal representations by error propagation, In D. E. Rumelhart, and J. L. McClelland, P*arallel Distributed Processing*, Vol. I Foundations, MIT Press, Cambridge, MA, 1986, pp. 318-362.

[20] M. Riedmiller and H. Braun, A direct adaptive method for faster backpropagation learning: the RPROP algorithm, *Proceedings of the IEEE International Conference on Neural Networks*, 1993, pp.586-591.

[21] S. E. Fahlman, An empirical study of learning speed in backpropagation networks, Technical report, CMU-CS-88-162, Carnegie-Mellon University, 1988.

[22] T. Tollenaere, Supersab: Fast adaptive backpropagation with good scaling properties, *Neural Networks*, 3(5), 1990, pp. 561-573.

[23] Jim. Torresen, and Olav. Landsvek, A review of parallel implementations of backpropagation neural networks, In N. Sundararajan, and P. Saratchandran, *Parallel Architectures for Artificial Neural Networks: Paradigms and Implementations*, IEEE Computer Society Press, Los Alamitos, California, 1998, pp. 25-63.

[24] L. Prechelt, PROBEN1: A set of neural network benchmark problems and benchmarking rules, Technical Report 21/94, Department of Informatics, University of Karlsruhe, Germany, 1994.

[25] R. O. Duda, and P.E. Hart, Pattern Classification and Scene Analysis, New York: Academic Express, 1973.

[26] S. Bernhard and J. Smola Alexander, Learning with kernels, Cambridge, Massachusetts: The MIT Press, 2002.

[27] J. Lampinen and A. Vehtari, Bayesian approach for neural networks, Neural Networks, April 2001, pp. 7-24.

[28] D. J. C. Mackay, A practical bayesian framework for backpropagation, Neural Computation, Vol. 4, 1992, pp. 448-472.

[29] D. J. C. Mackay, Bayesian methods for adaptive models, Ph.D. Thesis, California Institute of Technology, 1991.