Constrained-Storage Variable-Branch Neural Tree for Classification

Shiueng-Bien Yang
Department of Digital Content of Application and Management
Wenzao Ursuline University of Languages
900 Mintsu 1st Road Kaohsing 807, Taiwan.
Tel : 886-7-3426031
E-mail: 98010@mail.wzu.edu.tw

ABSTRACT

In this study, the constrained-storage variable-branch neural tree (CSVBNT) is proposed for pattern classification. In the CSVBNT, each internal node is designed as a single layer neural network (SLNN) that is used to classify the input samples. The genetic algorithm (GA) is proposed to search for the proper number of output nodes in the output layer of the SLNN. Furthermore, the growing method is proposed to determine which node has the highest priority to split in the CSVBNT because of storage constraint. The growing method selects a node to split in the CSVBNT according to the classification error rate and computing complexity of the CSVBNT. In the experiments, CSVBNT has lower classification error rate than other NTs when they have the same computing time.

***Key words*:** Neural trees, neural network, genetic algorithm.

I. INTRODUCTION

Decision trees (DT) ([1] - [6]) are commonly used techniques in machine learning, recognition and classification systems. The advantage of DTs is that decision nodes are easily designed and have low computing complexity. However, the disadvantage of DTs is that the classification error rate is high when a node is not a pure class. Therefore, large DTs are preferred to reduce the classification error rate, and then the DTs become deep trees and low performance. Neural trees (NT) ([7]–[11]) provide a solution for combining both decision trees and neural networks (NN), thus offering the advantages of both DTs and NNs. Recently, NTs have been applied to the recognition of character sets [12], human faces [13], range images [14], large pattern sets [15] and complex scenes [16].

Recently, the state-of-the-art NTs have been proposed. A neural tree with multi-layer perceptron (MLP) at the internal nodes was proposed by Guo and Gelfand [17]. In [17], the experimental results show that the NT with MLPs allows that the mode uses a smaller number of nodes and leaves. However, the disadvantage of NTs with MLPs is that the computing complexity is increased, since large number of parameters to be tuned and higher risk of overfitting. In [18], the adaptive high order neural tree (AHNT) is proposed. Nodes are high-order perceptrons (HOP) [19] whose order depends on the variation of the global error rate. First-order nodes divide the input space with hyperplanes, while HOPs divide the input space arbitrarily. The drawback of the AHNT is increased complexity and, thus, higher computational cost. In [20], the MLP is used to design the neural network tree (NNTree). Instead of using information gain ratio as splitting criterion, a new criterion is introduced for NNTree design. The new criterion captures well the goal of reducing the classification error rate. However, the main drawback is the necessity of an ad hoc definition of some parameters for each context and training set, such as the number of hidden layers, number of nodes for each layer in the neural networks. Although the NTs with MLPs previously described allow to divide the input space with arbitrary hypersurfaces, the computing complexity of a MLP is about several times the computing complexity of the single-layer neural network (SLNN). In [21], the single-layer perceptron tree was presented. In [22], Sirat and Nadal presented a binary structure of NT by using the SLNNs to solve two-class problems. Foresti and Micheloni [23] proposed a generalized neural tree (GNT), in which each node of the GNT is composed of two parts: the SLNN, and the normalizer. The learning rule is calculated by minimizing a cost function which represents a measure of the overall classification error of the GNT. In 2012, the balanced neural tree (BNT) was proposed to reduce tree size and improve classification with respect to classical neural tree [24]. The SLNN is also used to design each node in the BNT.

However, there are two common drawbacks to these state-of-the-art NTs, including the AHNT, NNTree, GNT and BNT. First, the growth strategies of these NTs only consider how to reduce the classification error rate, but do not consider how to reduce the computing complexity. Then, each NT is designed as a deep and large tree-structured NT to reduce the classification error rate, which results in the need for greater computing complexity. Therefore, an optimal NT must take into account how to reduce both the classification error rate and computing complexity when the depth or the number of nodes is limited in NT. The next drawback of these existing NTs is that the number of output nodes in the neural network (NN) is set to two (such as BNT) or the number of classes needed to be distinguished (such as GNT, AHNT and NNTree). In many cases, the number of output nodes in the output layer of NN is set to the number of classes needed to be distinguished, is not always the best stretgy to design the NT. If the number of output node in NN is too small, the samples belonging to different classes may be classified to the same output node in NN, and then a narrower and deeper NT will be generated. Otherwise, the computing complexity of NN is increased when the number of output node in NN is too many. Therefore, finding the proper number of output nodes in NN is an important research work in this study.

Weight assignment is one of the most important factors during NN design. Basically, the weights are controlled by both network architecture and the parameters of the learning algorithm. In addition, using several layers and nodes in hidden layers causes the network to be much more complex. Other NN parameters such as inputs, the number of hidden layers and their nodes, the number of memory taps and the learning rates also affect NN performance. Genetic algorithm (GA) ([25]-[34]) has been a popular approach to aiding neural network learning. In [30], GA is applied to the design of both NN structure evolution and weight adaptation. Mahmoudabadi et al. optimized an NN with the Levenberg-Marquardt (LM) and GA in order to improve its performance for grade estimation purposes [31]. Samanta et al. applied simulated annealing for NN training [32]. Chatterjee et al. applied GA in NNs, and showed that this combination can result in better NN performance [33]. Tahmasebi and Hezarkhani used GA to optimize NN parameters and topology, and obtained improved results [34]. However, the above GAs assume that the number of output nodes in an NN is known and can be set by the user in advance. This study proposes the CSVBNT which, unlike previously applied GAs, is able to automatically generate the proper number of output nodes in an NN. That is, when using the proposed GA, the user does not need to set the number of output nodes of an NN in advance.

The contributions of this study are summarized as follows:

(1) This study proposes the CSVBNT. To improve the first drawback of existing NTs, described above, the CSVBNT is designed to be optimized according to both the classification error rate and computing complexity of the CSVBNT. To fairly compare the performance of the CSVBNT and other NTs, the storage (i.e. the number of nodes in the NTs) is constrained. Because of the storage limitation, determining which node has the highest priority to split in the CSVBNT is an important design issue. The experiments conducted in this research demonstrate that the CSVBNT has a lower classification error rate than other existing NTs when they have the same computing time.

(2) To improve the second drawback of the existing NTs described above, GA is proposed to design the SLNN for each internal node in the CSVBNT. The proposed GA has the ability to automatically generate the weights and determine the proper number of output nodes in the SLNN according to both the classification error rate and computing complexity of the CSVBNT. Then, the CSVBNT tends to be optimized.

The remainder of this paper is organized as follows. The concept design of our proposed methods is described in Section II, and the design of CSVBNT with the storage constraint is presented in Section III. Section IV presents the design of the GA. The experiments are described in Section V, and Section VI presents the conclusions.

II. CONCEPT DESIGN OF OUR PROPOSED METHODS

In order to understand the entire design of the CSVBNT, the following describes the concept design of CSVBNT. The CSVBNT is a tree-structured classifier, in which each internal node is designed as an SLNN node, and each leaf node denotes the output space. Notably, the SLNN is a fully connected single-layer NN and each output node in the output layer of SLNN represents a branch in the CSVBNT. The two main contributions of the CSVBNT are described below:

(1) The growth strategy of the CSVBNT is designed based on both the classification error rate and computing complexity, and then the CSVBNT tends to be optimized. Figure 1 shows an example of

the performance of two NTs, NT_1 and NT_2. In Fig. 1, if the depth of the NT is increased, the average classification error rate is reduced and the average computation time is increased. From Fig. 1, although the same classification error rate e_1 can be achieved when NT_1 and NT_2 have sufficient depth, NT_1 still has better performance than NT_2. This is because the classification error rate of NT_1 is smaller than that of NT_2 ($e_2 < e_3$) when they have the same computing time, t. That is, NT_1 outperforms NT_2 when the depth or the number of nodes is limited. This situation means that if the storage space (i.e. the number of internal nodes in the NT) is limited, both the classification error rate and computing complexity must be used to identify the best NT. Therefore, the growth strategy of the CSVBNT takes into account how to reduce both the classification error rate and computing complexity, rather than the growth strategies of other NTs that only emphasize reducing the classification error rate.

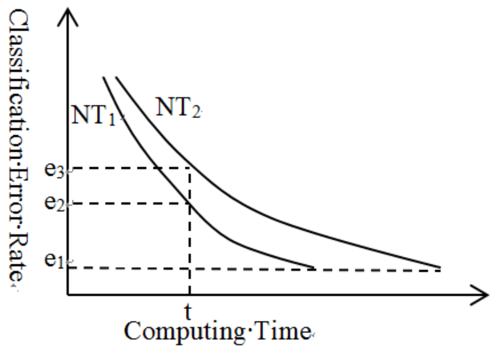
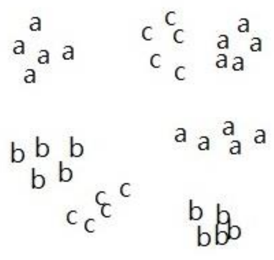
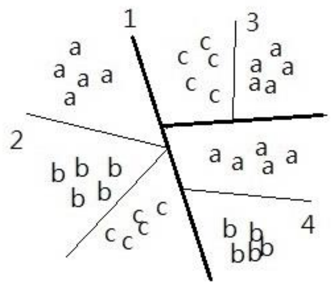


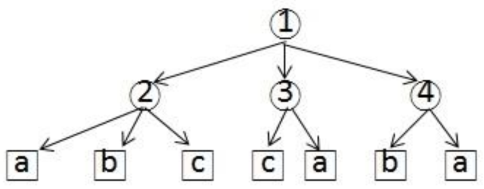
Figure 1. An example to illustrate two NTs.

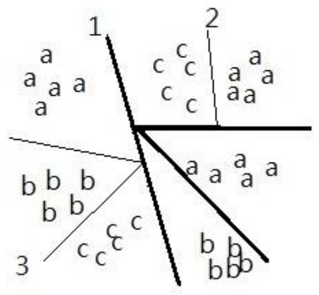
(2) The next contributions of the CSVBNT is that the proposed GA is enable to automatically search for the proper number of output nodes for each SLNN according to both the classification error rate and computing complexity of the CSVBNT. In the existing NTs, the number of output nodes in the output layer of NN is usually set to the number of classes needed to be distinguished, is not always the best growth stretgy to design the NT. Figure 2 shows a simple example to illustrate this situation. Figure 2(a) shows the original data set consisting of three classes of samples, Fig. 2(b) shows the partitions of the data set by the existing NTs, and Fig. 2(c) shows the corresponding tree classifier. Initially, the data set includes the three classes of samples (“a”, “b” and “c”) in Fig. 2(a). Since the data set has three classes of samples, the NN contained in the node 1 is then designed to have three

output nodes in Fig. 2(c). Similarly, node 3 in Fig. 2(c) consists of two classes of samples, “a” and “c”, and the NN contained in the node 3 is designed to have two output nodes to distinguish the two classes of samples. Therefore, an input sample classified as shown in Fig. 2(c) must be calculated with an average of two NNs. However, Fig. 2(d) shows another partition of data set, and Fig. 2(e) shows the corresponding NT that the node 1 is designed to have four output nodes. In Fig. 2(e), the classification of the input sample requires an average of 1.7 NNs (< 2 NNs). Thus, the NT shown in Fig. 2(e) than the NT shown in Fig. 2(c) has a shorter computing time when they have the same classification error rate. That is, the NN contained in the node 1 tends to be designed with four output nodes instead of three output nodes. Since the proposed GA can find the proper number of output nodes for the SLNN according to both the classification error rate and computing complexity of the CSVBNT, the CSVBNT becomes a variable-branch tree classifier that tends to be optimized.


(a) Original data set.


(b) Partition of the data set by the existing NTs.


(c) The existing NTs.


(d) Other partition of data set.

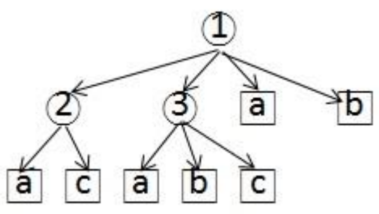
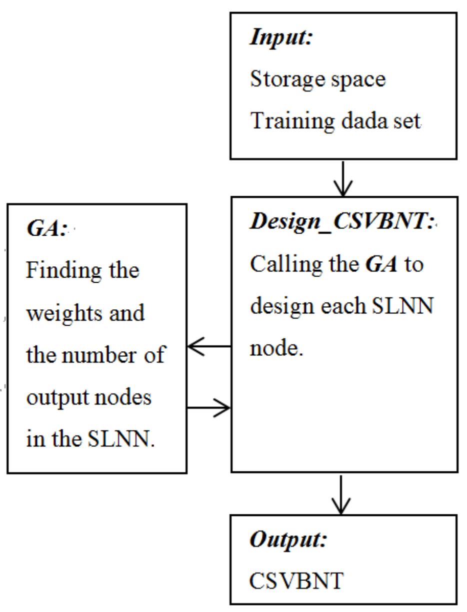

(e) Other better NT.

Figure 2. An example to illustrate the NTs.

The following describes the entire design flow of CSVBNT. In the training phase, the Design_CSVBNT algorithm described in Section III is proposed to design the CSVBNT. Figure 3(a) shows the flow chart of the Design_CSVBNT. In Fig. 3(a), the Design_CSVBNT algorithm is applied to design the CSVBNT by employing the GA described in Section IV to design the SLNNs in the CSVBNT. In the Design_CSVBNT, the number of internal nodes (SLNN nodes) denotes the storage space required to store the CSVBNT. The Design_CSVBNT is continued to design the CSVBNT from top to bottom until the storage space is reached. After the Design_CSVBNT is finished in the training phase, the CSVBNT can be obtained.

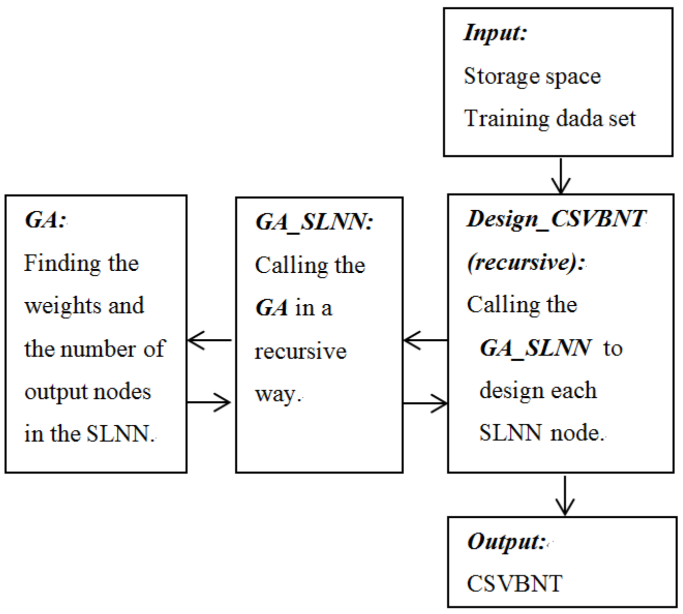
Furthermore, the Design_CSVBNT(recursive) algorithm is proposed to improve the efficiency of Design_CSVBNT in finding the best solution for the GA. The Design_CSVBNT(recursive) is a recursive version of Design_CSVBNT, and they all use the same GA to design the SLNNs in the CSVBNT. Figure 3(b) presents the flow chart of the Design_CSVBNT(recursive). In Fig. 3(b), the GA_SLNN is used to employ the GA in a recursive way. In the study, the Design_CSVBNT(recursive) instead of the Design_CSVBNT is used to generate the CSVBNT.

In the testing phase, the Test_CSVBNT algorithm described in Section III is used to explain how the input sample travels to the CSVBNT and derives the final classification result. Fig. 3(c) shows an example of the CSVBNT. In Fig. 3(c), the internal nodes, *A*, *B* and *C*, are designed as SLNNs, and the leaf nodes, *L1*, *L2*, *L3*, *L4*, *L5* and *L6*, are used to position the output space. Each output node in the SLNN has a related branch in the CSVBNT. In Fig 3(c), node *A* consists of three branches because the SLNN_A consists of three output nodes in the output layer. Also, SLNN_B and SLNN_C consist of two and three output nodes, respectively. Let the input sample be input to node *A* and classified to the output node, O_3, in the SLNN_A. The input sample then traces the related branch of O_3 and reaches node *C*. Similarly, the input sample continues to be classified in the SLNN_C, and reaches node L_4. Finally, the input sample is positioned in the same class as the arriving leaf node, L_4. The details of the testing algorithm, Test_CSVBNT, are described in Section III.



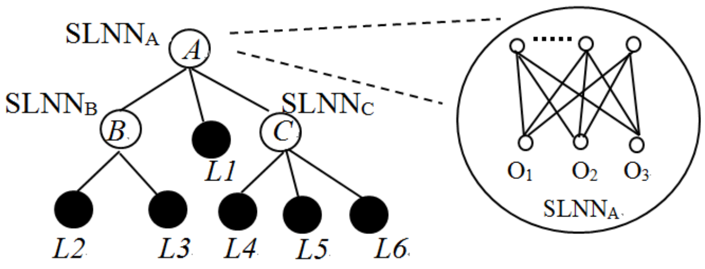
Input:
Storage space
Training dada set
Design_CSVBNT:
Calling the GA to design each SLNN node.
GA:
Finding the weights and the number of output nodes in the SLNN.
Output:
CSVBNT

(a) The flow chart of the Design_CSVBNT



Input:
Storage space
Training dada set
Design_CSVBNT (recursive):
Calling the GA_SLNN to design each SLNN node.
GA_SLNN:
Calling the GA in a recursive way.
GA:
Finding the weights and the number of output nodes in the SLNN.
Output:
CSVBNT

(b) The flow chart of the Design_CSVBNT(recursive)



(c) The example of CSVBNT (○: SLNN node ●: leaf node)

Figure 3. The concept design of the CSVBNT.

In order to enable readers to easily understand the proposed methods in this study, Table 1 lists the summary of important symbols used in Sections III, IV, and V.

Table 1. Summary of symbols used in this study

Symbols	Description
T	CSVBNT
SLNN_{t_i}	SLNN designed for the internal node, t_j
$V(t_j)$	computing complexity of the leaf node, t_j (Eq. (1))
$CC(t_i)$	computing complexity of SLNN_{t_i} (Eq. (2))
f	number of inputs in SLNN
r_i	number of outputs in SLNN_{t_i}
H	set of all leaf nodes in the CSVBNT, T
$P(t_j)$	probability of falling at node t_j (Eq. (3))
$V(T)$	computing complexity of the CSVBNT, T (Eq. (3))
$Class(X_k)$	class of training sample, X_k
C_j	j'th cluster
S_j	center of cluster C_j
m_{t_j}	number of training samples contained in the node t_j
$P(C_j, X_k)$	probability that X_k is classified to C_j (Eq. (5))
M_j	output space of node t_j (Eq. (6))
$R(t_j)$	classification error rate of node t_j (Eq. (7))
$R(T)$	classification error rate of the CSVBNT, T (Eq. (8))
$\lambda(t_j)$	slope of the classification error rate and computing complexity for the leaf node, t_j (Eq. (9))
δ	storage space (Design_CSVBNT)
ϕ_1, ϕ_2, ϕ_M	variables used to record the range of solution space (Design_CSVBNT(recursive))
θ	minimum solution space size (Design_CSVBNT(recursive))
SLNN_L	SLNN generated in the left subspace (Design_CSVBNT(recursive))
SLNN_R	SLNN generated in the right subspace (Design_CSVBNT(recursive))
θ_j	j'th activation threshold
$w_{i,j}$	weight between the i'th input node and the j'th output node in the SLNN
O_j	value of the j'th output node in the SLNN (Eq. (11))
N	population size
SLNN_{R_q}	SLNN encoded by the q'th string, R_q
$\lambda_q(t)$	changes in both the computing time and the classification error rate of CSVBNT after the SLNN for node t is generated by the string, R_q (Eq. (13))
ε	random value used to change the weights in the mutation phase (Eq. (15))
$v(i, j)$	wavelet coefficient at location (i, j) in the subband (Eq. (17))
n	size of the subband (Eq. (16) and Eq. (17))
P_c	crossover rate
P_m	mutation rate

III. DESIGN OF THE CSVBNT

In the design of CSVBNT, both of the computing complexity and classification error rate of CSVBNT are emphasized to be as small as possible. Before the design of CSVBNT is described, both of the computing complexity and classification error rate of CSVBNT are defined in the following.

The computing complexity of CSVBNT is defined as follows. Let *T* denote the CSVBNT, and let *H* denote the set of all leaf nodes in *T*, including leaf node t_j. Let $L(t_j)$ denote the set of internal nodes that are required to be calculated when the input sample travels the CSVBNT from the root node to the leaf node t_j. Then, the computing complexity of the leaf node t_j, $V(t_j)$, is defined as

$$V(t_j) = \sum_{t_i \in \mathrm{L}(t_j)} CC(t_i) \tag{1}$$

,where $CC(t_i)$ denote the computing complexity of SLNN_{t_i} for the internal node, t_i. Let the SLNN_{t_i} contain *f* inputs and r_i output nodes. Then, the $CC(t_i)$, is defined as

$$CC(t_i) = fr_i \,. \tag{2}$$

Let $P(t_j)$ represent the probability on the training samples in the leaf node t_j. The computing complexity of *T*, *V*(*T*), is defined as

$$V(T) = \sum_{t_j \in H} P(t_j) V(t_j) \tag{3}$$

The classification error rate of CSVBNT is described as follows. Let the leaf node t_j contain m_{t_j} training samples, $X_k = (x_k^1, x_k^2, ..., x_k^f) \in R^f$, for $1 \le k \le m_{t_j}$, and let $Class(X_k)$ denote the class of training sample, X_k. Let $C_j = \{(X_k, Class(X_k)) \mid 1 \le k \le m_{t_j}\}$ be a cluster that collects the training samples contained in the leaf node t_j. Let S_j be the center of cluster C_j. The center S_j of cluster C_j is defined as follows.

$$S_j = \frac{\sum_{k=1}^{m_{t_j}} X_k}{m_{t_j}} \,. \tag{4}$$

Thus, the probability that X_k is classified to C_j, $P(C_j, X_k)$, is defined as

$$P(C_j, X_K) = \frac{1}{\sum_{t_h \in H} [\frac{\|S_j - X_K\|}{\|S_h - X_K\|}]}, \tag{5}$$

where $\|\bullet\|$ denotes the Euclidean distance. Notably, $P(C_j, X_k)$ satisfies the constraints, $0 \le P(C_j, X_k) \le 1$ and $\sum_{t_h \in H} P(C_h, X_k) = 1$. Then, the representative of t_j positioned in the output space is given as

$$M_j = \frac{\sum_{\substack{(X_k, Class(X_k)) \in C_j \\ 1 \le k \le m_{t_j}}} P(C_j, X_k) Class(X_k)}{\sum_{\substack{(X_k, Class(X_k)) \in C_j \\ 1 \le k \le m_{t_j}}} P(C_j, X_k)}. \tag{6}$$

Then, the classification error rate of leaf node t_j, $R(t_j)$, is thus defined as

$$R(t_j) = \sum_{\substack{(X_k, Class(X_k)) \in C_j \\ 1 \le k \le m_{t_j}}} P(C_j, X_k)(Class(X_k) - M_j)^2. \tag{7}$$

Finally, the classification error rate of *T*, *R*(*T*), is defined as

$$R(T) = \sum_{t_j \in H} P(t_j) R(t_j). \tag{8}$$

In the design of CSVBNT, how to grow the tree structure of CSVBNT is described as follows. The growing method of CSVBNT selects the best leaf node to split at a time. That is, the GA described in Section III attempts to split each leaf node, and finds the best leaf node to branch in the CSVBNT. The following explains how to determine which one is the best leaf node. Let the leaf node t_j contain m_{t_j} training samples. If the leaf node t_j will be split in the CSVBNT, the GA is applied to these m_{t_j} samples to design an SLNN for the node t_j. Each output node in the output layer of the SLNN dnotes the corresponding child node of node t_j. Let the tree T_{t_j} indicate the tree *T* after r_j child nodes, $t_{j,1}, t_{j,2}, ..., t_{j,r_j}$, of t_j are generated by the GA. Then, the slope of the classification error rate and computing complexity for the leaf node, t_j, between *T* and T_{t_j} is:

$$\lambda(t_j) = \frac{\Delta R}{\Delta V} = \frac{R(T) - R(T_{t_j})}{V(T_{t_j}) - V(T)} \quad . \tag{9}$$

In the growing method of CSVBNT, the best leaf node is defined as the node with the largest value in Eq. (9). Therefore, let the set *H* consist of the leaf nodes in *T*. After each leaf node in *H* is designed as an SLNN, we can obtain

$$u = \arg \max_{t_j \in H} \lambda(t_j) \, . \tag{10}$$

Then, the node t_u is the best leaf node that has the highest priority to be split in *T*. Figure 4 shows an example to illustrate the growing method of CSVBNT. In Fig. 4, T_1 denotes the CSVBNT after the leaf node t_1 is split in *T*, and T_2 denotes the CSVBNT after the leaf node t_2 is split in *T*. The growing method selects one node at a time to split in *T*. From Fig. 4, T_2 is better than T_1 because the classification error rate of T_2 is smaller than that of T_1 when the computing time is the same. Thus, the aim of the design of CSVBNT is to maximize the slope of the classification error rate and computing complexity.

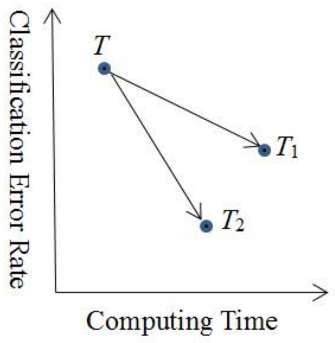


Figure 4. The relation between the classification error rate and computing time in CSVBNT.

The training and testing phases of CSVBNT are described below.

Training Phase:

In the training phase, two algorithms, Design_CSVBNT and Design_CSVBNT(recursive), are proposed to design the CSVBNT in this study. The Design_CSVBNT(recursive) is a recursive version of Design_CSVBNT. Before describing the design of Design_CSVBNT(recursive), the Design_CSVBNT is given as follows. Figure 3(a) also shows the flow chart of the Design_CSVBNT.

Notably, the storage space is defined as the number of internal nodes of the CSVBNT in the Design_CSVBNT.

```
Algorithm: Design_CSVBNT
Input:    The storage speac $\delta$ and the training data set.
Output:   The CSVBNT with the storage constraint.
Step 1.   Let the root node, $t$, of tree $T$ contain all of the training samples in the training data set. Set
          STORAGE= 0 and $H$={$t$}.
Step 2.   While    STORAGE $\leq$ $\delta$
    Step 2.1.   For each node $t_i \in H$ and $R(t_i) > 0$, perform the following.
        Step 2.1.1  The GA described in Section IV is applied to the samples contained in
                    the node $t_i$, and then the $\mathrm{SLNN}_{t_i}$ is produced for node $t_i$.
        Step 2.1.2  Let the $\mathrm{SLNN}_{t_i}$ contain $r_i$ output nodes in the output layer. Each
                    output node in the $\mathrm{SLNN}_{t_i}$ denotes a child node of node, $t_i$. Calculate
                    the value of $\lambda(t_i)$ as Eq. (9).
    Step 2.2.   Calculate $u = \arg\max_{t_i \in H} \lambda(t_i)$ as Eq. (10). The node, $t_u$, is the beast leaf noode that
                has the highest priority to be designed as an $\mathrm{SLNN}_{t_u}$, and then the new CSVBNT,
                $T_{t_u}$, is produced. Let there be $w$ child nodes for node $t_u$. Delete the node $t_u$ in the
                set of $H$ and add these $w$ new nodes to the set of $H$.
    Step 2.3.   Set $T$=$T_{t_u}$ and STORAGE=STORAGE+1.
Step 3.   Output the CSVBNT, $T$.
```

In Step 1, the root node *t* contains all the training samples. The set *H* represents a collection that contains all nodes that can be branched in the CSVBNT. Initially, set *H* contains only the root node *t*, in Step 1. In the first loop of Step 2, the set *H*={*t*} contains only the root node *t*, and then the root node *t*, is selected to be designed as an SLNN_t node by the GA algorithm in Step 2.1. In Step 2.2, assume that the root node t is split into three child nodes, t_1, t_2 and t_3. Then, the set $H = \{t_1, t_2, t_3\}$ can be obtained. In Step 2.3, the CSVBNT is built into a tree, T_t, that contains only one root node and three leaf nodes, and the STORAGE value is increased to one. Figure 5(a) shows the CSVBNT, T_t, after the first loop of Step 2 is complete. In the second loop of Step 2, the GA algorithm is applied to nodes t_1, t_2 and t_3, contained in H, then the three SLNN nodes SLNN_{t_1}, SLNN_{t_2} and SLNN_{t_3}, and three values, $\lambda(t_1)$, $\lambda(t_2)$ and $\lambda(t_3)$, are produced in Step 2.1. In Step 2.2, the best node is the one with the maximal value of λ. Assume that the value of $\lambda(t_1)$ is larger than both values of $\lambda(t_2)$ and $\lambda(t_3)$. Node t_1 is selected to be designed as the SLNN_{t_1} built into the CSVBNT. Assume that node t_1 contains two child nodes, t_{11} and t_{12}. Then, set H is updated to $\{t_{11}, t_{12}, t_2, t_3\}$. In Step 2.3, the STORAGE value is increased to two, and the new CSVBNT tree, T_{t_1}, replaces tree T_t. Figure 5(b) shows the CSVBNT, T_{t_1}, after the second loop of Step 2 is complete. In Step 2 of each loop, the best node is selected from set H to be designed as the SLNN node in the CSVBNT, and this step is repeated until the storage space limit is reached.

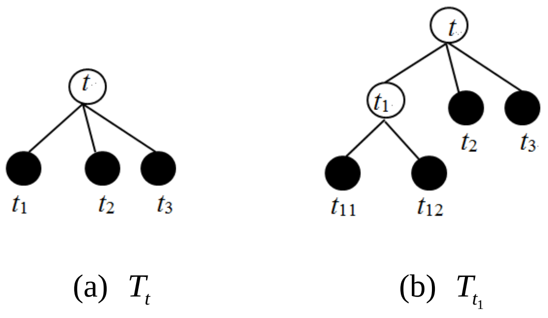
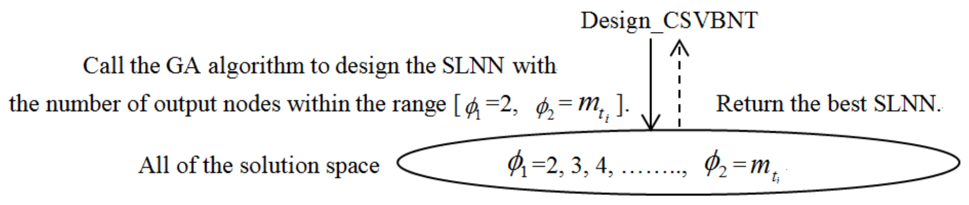


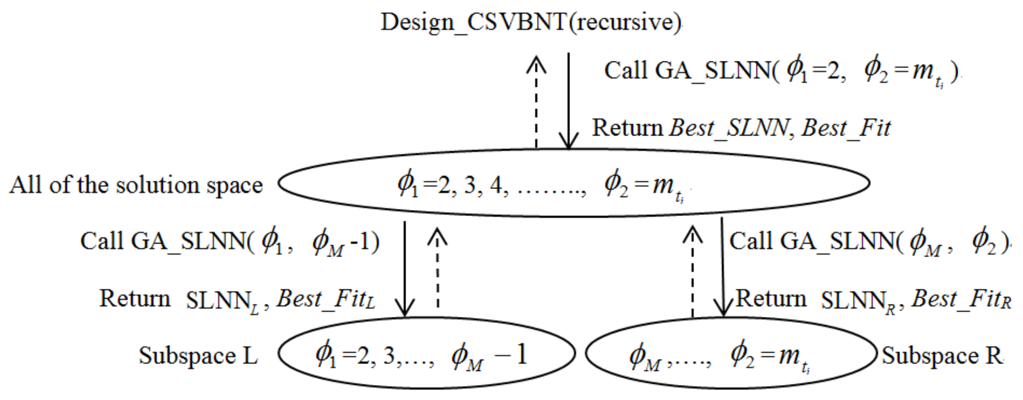
Figure 5. The example illustrates the Design_CSVBNT (○: SLNN node ●: leaf node).

In this study, the Design_CSVBNT(recursive), is also proposed to improve the efficiency of the Design_CSVBNT. The design of Design_CSVBNT(recursive) is described as follows. Figure 6 shows the difference in calling GA to find the best SLNN between Design_CSVBNT and Design_CSVBNT(recursive). In Step 2.1.1 of the Design_CSVBNT algorithm, the number of samples m_{t_i} contained in node t_i indicates that that number of output nodes in the SLNN_{t_i} can be up to m_{t_i}.

If the value of m_{t_i} is large, the GA should spend more time searching for the proper number of output nodes in the SLNN_{t_i} from m_{t_i} possible solutions, as shown in Fig. 6(a). To avoid the GA having to search a large solution space to find the solution, the overall solution space can be divided into two or more smaller subspaces in a recursive way, and then the GA can efficiently find the best solution in the smaller subspaces. Therefore, the Design_CSVBNT algorithm can be rewritten as a recursive version, namely Design_CSVBNT(recursive), which uses the divide-and-conquer strategy to design the SLNN by calling the recursive algorithm GA_SLNN, as shown in Fig. 6(b). Figure 3(b) also shows the flow chart of the Design_CSVBNT(recursive).


Design_CSVBNT
Call the GA algorithm to design the SLNN with the number of output nodes within the range [ϕ_1=2, $\phi_2 = m_{t_i}$].
Return the best SLNN.
All of the solution space
ϕ_1=2, 3, 4, ……., $\phi_2 = m_{t_i}$

(a) The Design_CSVBNT


Design_CSVBNT(recursive)
Call GA_SLNN(ϕ_1=2, $\phi_2 = m_{t_i}$)
Return *Best_SLNN, Best_Fit*
All of the solution space
ϕ_1=2, 3, 4, ……., $\phi_2 = m_{t_i}$
Call GA_SLNN(ϕ_1, ϕ_M-1)
Return SLNN_L, *Best_FitL*
Call GA_SLNN(ϕ_M, ϕ_2)
Return SLNN_R, *Best_FitR*
Subspace L
ϕ_1=2, 3,…, $\phi_M - 1$
ϕ_M,…, $\phi_2 = m_{t_i}$
Subspace R

(b) The Design_CSVBN(recursive)

Figure 6. The difference in calling GA to find the best SLNN between Design_CSVBNT and Design_CSVBNT(recursive). ($\longrightarrow$: Call, $\dashrightarrow$: Return)

The Design_CSVBNT(recursive) algorithm is described as follows.

Algorithm: Design_CSVBNT(recursive)

Input: The storage speac δ and the training data set.

Output: The CSVBNT with the storage constraint.

```
Step 1.  Let the root node, $t$, of tree $T$ contain all of the training samples in the training data set. Set
         STORAGE= 0 and $H=\{t\}$.
Step 2.  **While**   STORAGE $\leq \delta$
    Step 2.1.  For each node $t_i$ in $H$ and $R(t_i) > 0$, perform the following.
        Step 2.1.1  Let $m_{t_i}$ be the number of samples contained in the node $t_i$. Call the
                    **GA_SLNN**($\phi_1$=2, $\phi_2 = m_{t_i}$) to obtain the $\mathrm{SLNN}_{t_i}$ for the leaf node $t_i$.
        Step 2.1.2  Let the $\mathrm{SLNN}_{t_i}$ contain $r_i$ output nodes in the output layer. Each
                    output node in the $\mathrm{SLNN}_{t_i}$ denotes a child node of the node, $t_i$.
                    Calculate the value of $\lambda(t_i)$ as Eq. (9).
    Step 2.2.  Calculate $k = \arg\max_{t_i \in H} \lambda(t_i)$ as Eq. (10). The node, $t_k$, is the beast leaf noode that
               has the highest priority to be designed as an $\mathrm{SLNN}_{t_k}$, and then the new CSVBNT,
               $T_{t_k}$, is produced. Let there be $w$ child nodes for the node $t_k$. Delete the node $t_k$ in
               the set of $H$ and add these $w$ new nodes to the set of $H$.
    Step 2.3.  Set $T= T_{t_k}$ and STORAGE=STORAGE+1.
Step 3.  Output the CSVBNT, $T$.
```

Algorithm: GA_SLNN (Input:ϕ_1,Input:ϕ_2)

```
Step 1.  If $\phi_2 - \phi_1 \leq \theta$, then GA is applied to these $m_{t_i}$ samples contained in the leaf node $t_i$, to
         design the $\mathrm{SLNN}_{t_i}$ whose number of output nodes in the output layer is within the range
         [$\phi_1$, $\phi_2$]. Set *Best_SLNN*=$\mathrm{SLNN}_{t_i}$ and *Best_Fit*= the value of the best fitness.
         **Return** both the *Best_SLNN* and *Best_Fit*. **End**.
Step 2.  If $\phi_2 - \phi_1 > \theta$, do the following.
    Step 2.1  Calculate $\phi_M = \left\lceil \frac{\phi_2 - \phi_1}{2} \right\rceil$.
```

```
        Step 2.2  Call **GA_SLNN**($\phi_1$, $\phi_M$-1). Then, both the $\mathrm{SLNN}_L$ and $Best\_Fit_L$ can be obtained.
        Step 2.3  Call **GA_SLNN**($\phi_M$, $\phi_2$). Then, both the $\mathrm{SLNN}_R$ and $Best\_Fit_R$ can be obtained.
Step 3. If $Best\_Fit_R > Best\_Fit_L$ Then $Best\_SLNN = \mathrm{SLNN}_R$ and $Best\_Fit = Best\_Fit_R$.
        Otherwise $Best\_SLNN = \mathrm{SLNN}_L$ and $Best\_Fit = Best\_Fit_L$.
        Return both the Best_SLNN and Best_Fit. End.
```

In Step 2.1.1 of Design_CSVBNT(recursive), GA_SLNN(ϕ_1=2, $\phi_2 = m_{t_i}$) is a recursive algorithm. After calling the GA_SLNN(ϕ_1=2, $\phi_2 = m_{t_i}$), the SLNN_{t_i} with the number of output nodes within the range [ϕ_1=2, $\phi_2 = m_{t_i}$] can be generated by the GA. Notably, two classes (ϕ_1=2) are the minimum number of classes that need to be distinguished, and the maximal number of output nodes in SLNN_{t_i} is equal to m_t ($\phi_2 = m_{t_i}$) such that each sample is regarded as the only member of its own class. Therefore, the SLNN_{t_i} generated by the GA still tends to be optimal because the GA has a global search within the range [2, m_t].

In Step 1 of the GA_SLNN, if the range of the solution space, [ϕ_1, ϕ_2], is small, the GA is directly used to design the SLNN with the number of output nodes in the output layer within the range [ϕ_1, ϕ_2]. Then, the GA_SLNN returns both the best SLNN and the maximal fitness value generated by the GA. In the Step 2 of the GA_SLNN, if the range of the solution space, [ϕ_1, ϕ_2], is large, the solution space, [ϕ_1, ϕ_2], is divided into two subspaces, [ϕ_1, ϕ_M-1] and [ϕ_M, ϕ_2], in a recursive way. That is, the GA is used to design the SLNNs in these two subspace, [ϕ_1, ϕ_M-1] and [ϕ_M, ϕ_2], instead of the whole solution space, [ϕ_1, ϕ_2]. In Step 3, the GA_SLNN returns both of the variables, *Best_SLNN* and *Best_Fit*, recording the best SLNN and the maximal value of the fitness generated by GA in these two subspaces, respectively.

Notably, in the GA_SLNN, the threshold θ is not a critical value for users. If the threshold θ is small, the overall solution space can be divided into smaller (or more) subspaces, and then the GA can efficiently find the best solution in the smaller subspaces. Otherwise, the solution space will be cut into larger (or fewer) subspaces. However, if the threshold θ is large enough, the Step 1 will only be performed in GA_SLNN, and then Design_CSVBNT(recursive) and Design_CSVBNT become the same algorithm.

After the training phase is finished, the CSVBNT can be obtained. In the CSVBNT, each leaf node is used to position the output class of the input samples. However, once there will be many different classes of training samples classified to the same leaf node in the training phase. Then, the output class representing a leaf node is defined as the class to which the largest number of training samples belongs in the leaf node.

Testing Phase:

The testing algorithm, Test_CSVBNT, is proposed to classify an input sample in the CSVBNT. Before the Test_CSVBNT is given, the following describes how to classify an input sample in the $SLNN_t$ for the internal node *t*. Let the $SLNN_t$ contain *f* inputs and r_t output nodes. The values of θ_j for $0<\theta_j<1$, $1\le j\le r_t$ denote the activation thresholds. The values of $w_{i,j}$ for $0<w_{i,j}<1$, $1\le i\le f$, and $1\le j\le r_t$, indicate the weights between the input and output layers. Let the sample, $X_k=(x_k^1,x_k^2,...,x_k^f)\in R^f$, be the input to the $SLNN_t$. In the $SLNN_t$, the value of the *j*th output node, O_j, is the sum of its weighted inputs as follows.

$$O_j=(\sum_{i=1}^{f}w_{ij}x_k^i)-\theta_j,\ \ 1\le j\le r_t \tag{11}$$

Let

$$u=\arg\max\{O_j \mid 1\le j\le r_t\}. \tag{12}$$

The input sample, X_k, is then classified to the *u*th output node in the $SLNN_t$.

The Test_CSVBNT algorithm is given as follows.

Algorithm: Test_CSVBNT

```
Input:   The input sample $X_k = (x_k^1, x_k^2, ..., x_k^f) \in R^f$ and tree classifier, CSVBNT.
Output:  The class of $X_k$.
Step 1.  Let $t$ be the root node of the CSVBNT.
Step 2.  While  $t$ is not a leaf node
         Step 2.1. The input sample, $X_k$, is the input to the $SLNN_t$. Calculate the values, $O_j$, for
                   $1 \le j \le r_t$, as Eq. (11).
         Step 2.2. Calculate $u = \arg\max\{O_j \mid 1 \le j \le r_t\}$ as Eq. (12).
         Step 2.3. The input sample, $X_k$, is classified to the uth output node in the $SLNN_t$. Then, the
                   input sample, $X_k$, towards the corresponding child node, $t_u$, of the node, $t$, in the
                   CSVBNT.
    Step 2.3.  Set $t = t_u$
Step 3.  Output the representative class of node $t$.
```

IV. DESIGN OF THE GA

There are two design issues for the GA. First, the GA searches for the weights in the SLNN. Next, the GA can automatically search for the proper number of output nodes in the output layer of the SLNN based on the classification error rate and computing complexity of CSVBNT. Let *T* be the CSVBNT, and let *t* be the node that contains m_t input samples in *T*. The main goal of GA is to design the SLNN for node *t*. Each output node in the output layer of the SLNN is designed as a corresponding child node of node *t*, in the CSVBNT. The GA is designed based on the genetic algorithm, which includes the initialization step and three phases: reproduction, crossover, and mutation. They are described as follows.

Initialization

In the initialization step of GA, a population of N strings, $R_1, R_2, ..., R_N$, is randomly generated. The length of each string is set to be smaller than, or equal to, the value of m_t, i.e. the length of each string is variant in GA. That is, the number of child nodes of node t generated by the GA is within the range [2, m_t]. Let the string R_q encode the SLNN_{R_q}, which has f inputs and r_{R_q} output nodes, for $1 \le q \le N$. The values of θ_j denote the j'th activation thresholds for $0 < \theta_j < 1$, $1 \le j \le r_{R_q}$. The values of $w_{i,j}$ indicate the weights between the input and output layers for $0 < w_{i,j} < 1$, $1 \le i \le f$ and $1 \le j \le r_{R_q}$. The string R_q then encodes the SLNN_{R_q}, as follows.

R_q=(*O_node*$_q$(1),*O_node*$_q$(2),…, *O_node*$_q$(r_{R_q}))

, where *O_node*$_q$(j)=($\theta_j, w_{11}, \ldots, w_{fj}$), for $1 \le j \le r_{R_q}$.

The following is an example of the initialization step. Let there be two elements (f=2) for each input sample, and let r_{R_q}=3 be a random value generated within [2, m_t]. R_q then encodes the solution as follows.

R_q=($\theta_1, w_{11}, w_{21}, \theta_2, w_{12}, w_{22}, \theta_3, w_{13}, w_{23}$)

=(*O_node*$_q$(1), *O_node*$_q$(2), *O_node*$_q$(3))

,where *O_node*$_q$(1)=(θ_1, w_{11}, w_{21}), *O_node*$_q$(2)=(θ_2, w_{12}, w_{22}), and *O_node*$_q$(3)=(θ_3, w_{13}, w_{23}). Figure. 7 also shows the corresponding SLNN_{R_q} encoded by the string R_q.

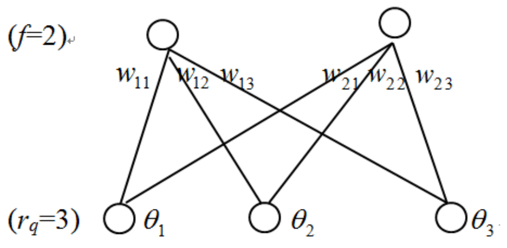


Figure 7. The corresponding SLNN_{R_q} encoded by the string R_q.

Reproduction:

In the reproduction phase, the definition of fitness depends on both the classification error rate and computing complexity of the CSVBNT. The following describes the definition of fitness for the string R_q. Before the fitness of string R_q is defined, the following describes how to classify a training sample in the $SLNN_{R_q}$. Let $X_k = (x_k^1, x_k^2, ..., x_k^f) \in R^f$ be a sample contained in node *t*. In the SLNN_{R_q}, the values, $O_j = (\sum_{i=1}^{f} w_{ij} x_k^i) - \theta_j$, for $1 \le j \le r_{R_q}$, are calculated according to Eq. (11), and the value of $u = \arg\max\{O_j \mid 1 \le j \le r_{R_q}\}$ is calculated via Eq. (12). Sample X_k is then classified to the *u*th output node in the SLNN_{R_q}. After all of these m_t training samples contained in node *t* have been classified to these r_{R_q} output nodes, the training samples classified to the same output nodes can be collected to be designed as child nodes of *t*. That is, r_{R_q} child nodes $t_1, t_2, ..., t_{R_q}$ of node *t* are generated based on these r_{R_q} output nodes in the SLNN_{R_q}. Let the tree T' indicate the tree *T* after the r_{R_q} child nodes of node t have been generated. The fitness function of string R_q is then defined as:,

$$Fitness(R_q) = \lambda_q(t) \text{ , for } 1 \le q \le N, \tag{13}$$

where $\lambda_q(t)$ is defined in Eq. (9) as representative of changes in both the computing time and the classification error rate of CSVBNT.

After the fitness values of these *N* strings, $R_1, R_2, ..., R_N$, in the population have been obtained, the probability of each string being selected for the next generation can then be calculated as follows:

$$Prob(R_q) = \frac{\lambda_q(t)}{\sum_{l=1}^{N} \lambda_l(t)}, \text{ for } 1 \le q \le N. \tag{14}$$

Notably, $Prob(R_q)$ satisfies $0 < Prob(R_q) < 1$ for $1 \le q \le N$, and $\sum_{q=1}^{N} Prob(R_q) = 1$. In the reproduction phase, the reproduction operator selects *N* strings as the new population in the next generation according to the probability values, $Prob(R_q)$ for $1 \le q \le N$.

The following shows an example of the reproduction phase. Let there be five probability values, $Prob(R_1)$=0.2, $Prob(R_2)$=0.15, $Prob(R_3)$=0.1, $Prob(R_4)$=0.15 and $Prob(R_5)$=0.4, for five strings, R_q for $1 \le q \le 5$, in the population. Figure 8 shows the distribution of these five probability values in the range [0, 1]. In the reproduction phase, the reproduction operator generates five random values within [0, 1] such as 0.15, 0.3, 0.5, 0.7 and 0.9, to determine which strings are selected for the next generation. From Fig. 8, the reproduction operator selects R_1, R_2, R_4, R_5, and R_5, to be the new population in the next generation. Notably, the string R_5 is selected twice and R_3 is omitted. The meaning of the reproduction phase is that the string with higher fitness has a greater probability of being repeatedly selected and the strings with lower fitness may be removed in the next generation.

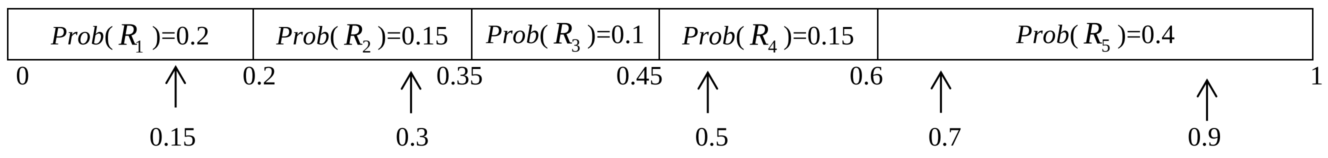


Figure 8. The distribution of five probability values in the range, [0. 1].

Crossover:

In the crossover phase, the crossover operator is applied to the population of *N* strings. Then, a pair of strings, R_x and R_y, is selected to do crossover operator. Then, two random integers, *a* and *b*, are generated to determine which pieces of the strings are to be interchanged. Notably, if the number of output nodes contained in the string is outside the range [2, m_t] after the crossover operator is completed, the two values, *a* and *b*, should be randomly generated again. After the crossover operator is finished, two new strings, $\hat{R}_x$ and $\hat{R}_y$, replace the strings, R_x and R_y, in the population. The significance of the crossover phase is that it exchanges the output nodes including the connected weights between different strings, to yield various neural networks.

The following shows an example of the crossover phase. Let there be two elements (*f*=2) for each sample. Let R_1 contain three output nodes, and let R_2 contain four output nodes. Then, after the crossover operator with two random integers (*a*=1 and *b*=2) is applied to the pair of strings, R_1 and R_2, two new strings, $\hat{R}_1$ and $\hat{R}_2$, are generated in the next generation.

$$R_1 = (O\_node_1(1), \overset{a=1}{\Big|} O\_node_1(2))$$

, where

$O\_node_1(1)=(\theta_1, w_{11}, w_{21})$,

$O\_node_1(2)=(\theta_2, w_{12}, w_{22})$.

$$R_2 = (O\_node_2(1), O\_node_2(2), \overset{b=2}{\Big|} O\_node_2(3))$$

, where

$O\_node_2(1)=(\theta_1, w_{11}, w_{21})$,

$O\_node_2(2)=(\theta_2, w_{12}, w_{22})$,

$O\_node_2(3)=(\theta_3, w_{13}, w_{23})$.

After the crossover operator is applied to R_1 and R_2, two new strings, $\hat{R}_1$ and $\hat{R}_2$, are generated as follows.

$\hat{R}_1 = (O\_node_2(1),\ O\_node_2(2),\ O\_node_1(2))$

$\hat{R}_2 = (O\_node_1(1), O\_node_2(3))$

Mutation:

In the mutation phase, the weights of the strings in the population are randomly chosen with a probability. Each chosen weight is added by the multiplication of the chosen weight and a random value (0< ε <1). The following shows an example of the mutation phase. Let R_1 be presented as follows:

$R_1 = (O\_node_1(1),\ O\_node_1(2))$,

where

$O\_node_1(1)=(\theta_1, w_{11}, w_{21})$,

$O\_node_1(2)=(\theta_2, w_{12}, w_{22})$.

If the weight w_{11} is chosen to perform the mutation, the new weight w'_{11} replaces weight w_{11} as follows:

$$w'_{11} = w_{11} \pm w_{11} * \varepsilon \, , \tag{15}$$

where the value, ε, is a random value within the range [0, 1]. After the mutation phase, the new string can be obtained and replace the original string.

The user may specify the number of generations over which to run in the GA. Suppose that the string $\hat{R}_x$ with the best fitness generates the SLNN with $\hat{r}_x$ output nodes. Then, $\hat{r}_x$ child nodes of node *t* are generated in *T*, according to these $\hat{r}_x$ output nodes contained in the SLNN.

V. EXPERIMENTS

A. Experimental setting

In the experiments, the Design_CSVBNT(recursive) algorithm is used to design the CSVBNT with the storage constraint. The proposed CSVBNT is compared with other NTs under the same number of internal nodes. A run of twenty times is carried out for accurately proposing the results of the CSVBNT and other NTs. In our proposed methods, the threshold, $\theta = \frac{m_{t_i}}{10}$, is applied to the GA_SLNN since the one-tenth size of the overall solution space is sufficiently small for the GA to efficiently find the best solution. Also, the parameters used in the GA are as follows: population of 300, crossover rate, *Pc* = 80%, and mutation rate, *Pm* = 5%. Five hundred generations are run in the GA, and the best solution is retained. All the experiments are carried out on personal computers.

Four data sets: speech, traffic sign images, natural images, and chessboard data sets are used to test the CSVBNT and other NTs in the experiments. These four data sets are described as follows.

(1) In the speech data set, the ISOLET database using the 26 letters of the English alphabet is used in the isolated word recognition test. The speech data set consists of 6240 utterances including 4000 training utterances and 2240 testing utterances, from 120 speakers. Each utterance is sampled at 16 kHz with a 16-bit resolution. A Hamming window of 20 ms with 50% overlap is used to process each utterance further by Fast Fourier Transform (FFT). Each utterance is divided into 15 Hamming windows, with each represented by 32 FFT coefficients; that is, each utterance consists of 480

features.

(2) The traffic sign images data set is obtained from the GTSRB database [35]. The training set consists of 5000 images, and the other 5000 images are used to test the methods in our experiments. All images belong to forty classes. The actual traffic sign is not always centered within the image; its bounding box is part of the annotations. We crop all the images and process them only within the bounding box, and resize them to achieve square bounding boxes. All traffic sign images are resized to 48x48 pixels.

(3) The natural images of 32x32 pixels are obtained from the CIFAR10 database [36]. The CIFAR10 database consists of ten classes of images, each with 5000 training images and 1000 testing images. Images vary greatly within each class. They are not necessarily centered, and may contain only parts of the object, and show different backgrounds.

(4) The symmetrically distributed four-class chessboard data set is used to test the CSVBNT and other current methods. Figure 4(a) shows the chessboard data set that consists of 400 patterns equally distributed among four classes. A five-fold cross-validation is performed, and average results are presented.

In both the traffic sign and natural images data sets, all color images should be transformed into the gray images in the experiments. Then, each image is divided into blocks of 16×16 pixels. Each block is then transformed by a Haar wavelet transform [37] to obtain four subbands. The mean values (*mv*) and standard deviations (*sd*) of the four subbands are calculated as follows:

$$mv = \frac{1}{n^2} \sum_{i,j=1}^{n} v(i, j), \tag{16}$$

$$sd = \sqrt{\frac{1}{n^2} \sum_{i,j=1}^{n} \left[v(i, j) - mv \right]^2}, \tag{17}$$

where *n* denotes the size of the subband, which is set to 8 in this experiment, and *v*(*i*, *j*) denotes the wavelet coefficient at location (*i*, *j*) in the subband. Therefore, each block, which contains four subbands, can be represented by a feature vector with eight values since each subband is associated with two values, *sd* and *mv*.

B. Performance of CSVBNT

Before testing the performance of CSVBNT, the sensitivity of these two parameters, Pc and Pm , in the GA is described as follows. If Pc is set to 50%, the GA requires twice the number of generations to get a similar or poor solution in four data sets, compared with when Pc is set to 80% or 90%. This is because the use of a too small Pc will affect the efficiency of the GA. Also, when the probability Pm is set to less than 10%, the GA can obtain similar results under the same number of generations. However, if Pm is set to 15%, the GA usually does not converge. This is because the strings are changed too much in the GA, so that the GA cannot converge to a solution. Therefore, both parameters, Pc = 80% and Pm = 5%, are applied to the GA in the experiments.

In Table 2, the different storages of δ are used to design the CSVBNT on four datasets. In this study, the storage is defined as the number of internal nodes in the CSVBNT. Notably, because each internal node contains an SLNN, the storage, δ, also denotes the total number of SLNNs dsigned in the CSVBNT. In Table 2, the "Num_O" denotes the average number of child nodes of an internal node (or the average number of output nodes in the output layer of an SLNN), the "DEP" denotes the average depth of the CSVBNT, and the "CER" denotes the average classification error rate on the training dataset. The classification error rate is defined as Eq. (8). When the value of δ is set to ∞, it represents that the storage space is unlimited, and then the CSVBNT is grown until the classification error rate is less than a small threshold ε ($\varepsilon = 2\%$), or the training error rate exhibits no obvious decrease.

In the experiments, the proposed CSVBNT is compared with four NTs: GNT [23] , AHNT [18], NNTree [20] and BNT [24]. To fairly compare our proposed CSVBNT and other NTs, the CSVBNT and other NTs have the same storage space (*i.e.,* the same number of internal nodes) in the experiment. In Table 3, the "TIME" denotes the average computing time for a testing sample, and the "CER" denotes the average classification error rate on the testing dataset. In Table 3, we observe that both of the AHNT and NNTree than the proposed CSVBNT have lower classification error rate when the storage space is the same. The reason is that the MLP allows to divide the input space with arbitrary hypersurfaces in the AHNT and NNTree. That is, if the distribution of samples contained in the node is complex (non-linear distribution), the node is preferred in designing an MLP, and then the classification error rate of both the AHNT and NNTree can be decreased. However, we also observe that both of the AHNT and NNTree using the MLPs than the CSVBNT using the SLNNs take more computing time when they have the same storage in Table 3. The reason is that the SLNN than the MLP has lower computing complexity. The computing complexity of an MLP is usually larger than twice the computing complexity of the SLNN.

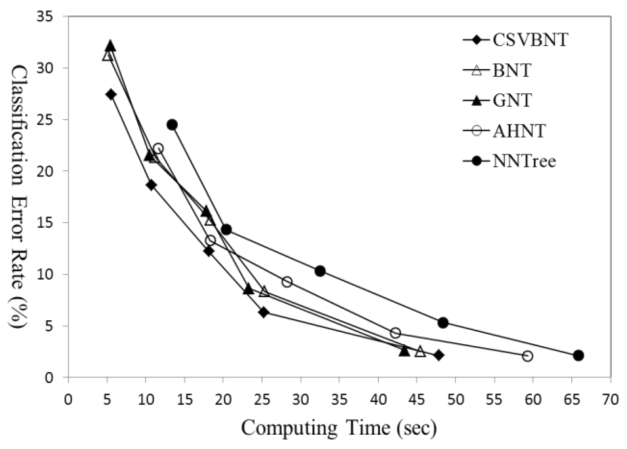
Figure 9 shows the experimental results proposed in Table 3, and Figure 10 shows the classification results obtained by the CSVBNT on the chessboard data set. From Fig. 9, we observe that the CSVBNT than other NTs has lower classification error rate when they have the same computing time. Two reasons are offered as follows. (1) The proposed growing strategy designs the CSVBNT according to the classification error rate and computing complexity of CSVBNT, while the other NTs including GNT, AHNT, NNTree and BNT, only consider the reduction of the classification error rate. (2) The GA is capable of searching for the proper number of output nodes in the SLNN according to the classification error arte and computing complexity of CSVBNT. Figures 1 and 2 has shown that the characteristics of the CSVBNT differ from those of other NTs.

Table 2. The efficiency of the CSVBNT on the training datasets.

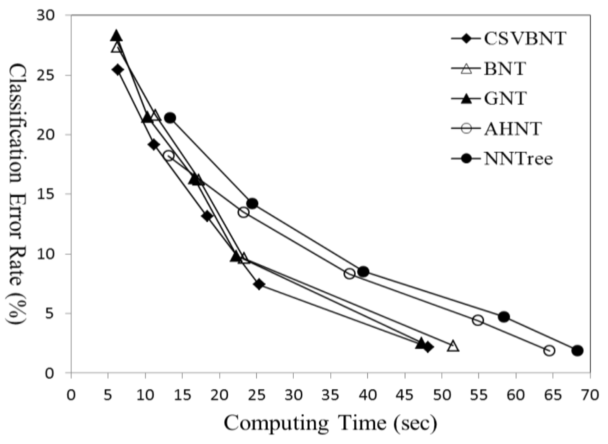
Data Sets	δ	**Num_O**	**DEP**	**CER (%)**
Speech	5	3.25	3.43	22.36
	8	3.18	5.25	12.37
	11	3.25	7.46	8.29
	14	3.25	8.35	4.21
	∞	3.33	14.25	1.52
Traffic sign images	5	5.35	3.43	21.42
	8	5.52	5.54	16.32
	11	5.53	6.34	8.74
	14	5.54	8.25	3.27
	∞	5.52	13.78	1.39
Natural images	5	2.44	3.36	25.43
	8	2.64	4.46	18.48
	11	2.52	6.64	13.35
	14	2.56	8.63	7.62
	∞	2.67	12.38	1.78
Chess-board	3	2.00	3.47	18.35
	6	2.00	4.53	13.48
	9	2.00	5.35	7.48
	12	2.00	7.45	3.49
	∞	2.00	10.58	1.92

Table 3. The performance the CSVBNT and other NTs on the testing datasets

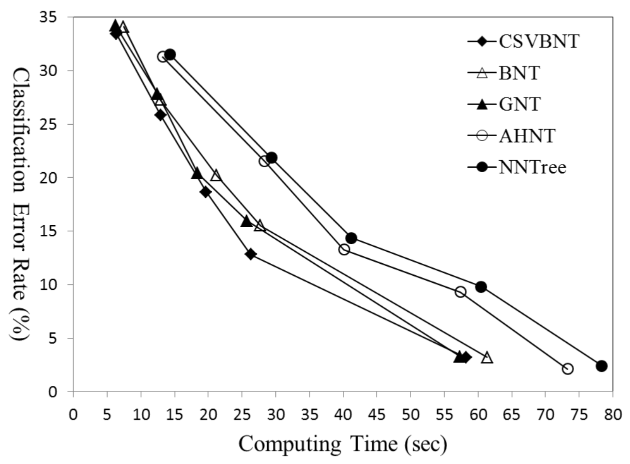
Data Sets	δ	**CSVBNT**		**GNT [23]**		**BNT [24]**		**NNTree [20]**		**AHNT [18]**	
		CER (%)	**TIME (sec)**	**CER (%)**	**TIME (sec)**	**CER (%)**	**TIME (sec)**	**CER (%)**	**TIME (sec)**	**CER (%)**	**TIME (sec)**
Speech	5	27.42	5.52	32.18	5.43	31.28	5.13	24.52	13.37	22.22	11.63
	8	18.64	10.72	21.52	10.52	21.35	11.18	14.32	20.38	13.26	18.34
	11	12.23	18.15	16.12	17.85	15.27	18.33	10.35	32.48	9.29	28.25
	14	6.32	25.22	8.62	23.32	8.31	25.36	5.33	48.37	4.3	42.23
	∞	2.12	47.85	2.62	43.43	2.53	45.43	2.11	65.83	2.08	59.32
Traffic sign images	5	25.42	6.33	28.32	6.13	27.33	6.25	21.42	13.31	18.24	13.13
	8	19.12	11.24	21.44	10.34	21.65	11.34	14.21	24.45	13.51	23.25
	11	13.14	18.37	16.32	16.64	16.21	17.24	8.53	39.42	8.33	37.52
	14	7.42	25.38	9.82	22.32	9.63	23.33	4.72	58.34	4.42	54.84
	∞	2.19	48.12	2.52	47.32	2.31	51.52	1.91	68.32	1.86	64.52
Natural images	5	33.42	6.38	34.22	6.22	34.12	7.42	31.51	14.32	31.31	13.22
	8	25.84	12.92	27.82	12.44	27.27	12.94	21.85	29.34	21.52	28.35
	11	18.65	19.66	20.42	18.42	20.22	21.23	14.36	41.23	13.26	40.13
	14	12.82	26.32	15.95	25.73	15.52	27.74	9.82	60.44	9.32	57.4
	∞	3.19	58.27	3.29	57.32	3.21	61.35	2.42	78.33	2.12	73.3
Chess-board	3	25.21	6.43	26.22	6.13	26.14	6.32	20.22	12.34	19.12	12.54
	6	18.54	11.29	20.34	11.12	20.18	11.52	14.82	28.48	14.28	26.38
	9	11.65	19.42	13.76	18.76	13.28	18.74	9.54	46.33	9.44	42.33
	12	6.46	25.34	8.33	24.14	8.12	24.21	5.61	58.21	5.31	53.29
	∞	2.92	54.28	2.94	53.63	2.72	55.32	2.25	79.54	2.15	77.24



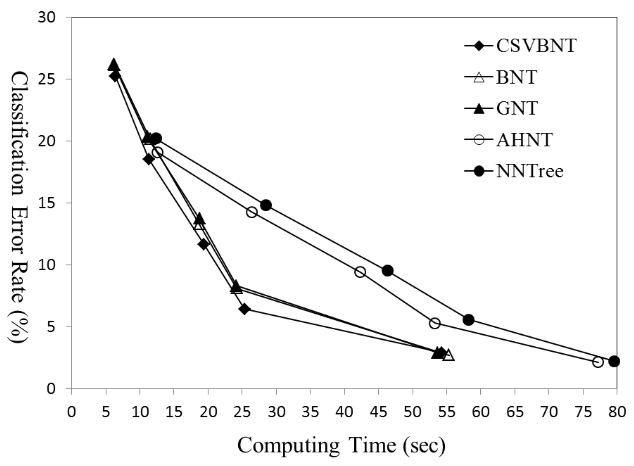
(a) Speech dataset



(b) Traffic sign images dataset

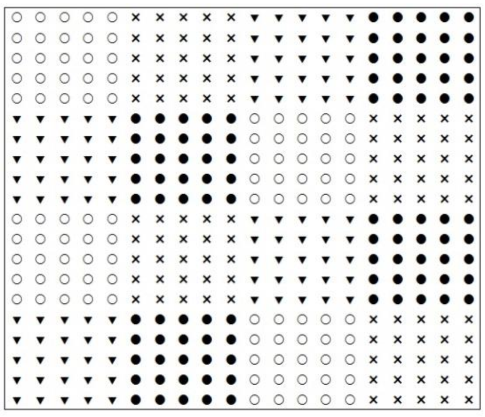


(c) Natural images dataset

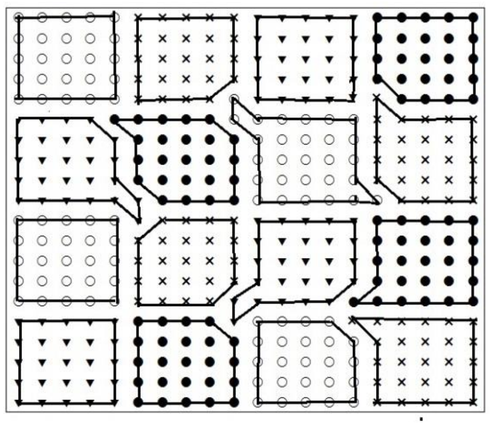


(d) Chessboard dataset

Figure 9. The performance of CSVBNT and other NTs.



(a) Chessboard data set



(b) The result obtained by the CSVBNT.

Figure 10. The classification result by the CSVBNT on the chessboard dataset.

VI. CONCLUSIONS

This study proposes the CSVBNT based on GA. The CSVBNT tends to be optimal because its design takes into account how to reduce both the classification error rate and computing complexity. The CSVBNT is also a variable-branch neural tree because the number of output nodes in the output layer of each SLNN node is automatically determined by the GA according to both the classification error rate and computing complexity of the CSVBNT. Furthermore, this study proposes Design_CSVBNT (recursive), which operates similarly to the divide-and-conquer strategy for efficient SLNN design. Design_CSVBNT(recursive) is able to determine which node has the highest priority to be selected to split in the CSVBNT under the storage constraint. The experiment results in this study demonstrate that the CSVBNT has lower classification error rate than existing NTs when they have the same computing time.

Disclosure of Potential Conflicts of Interest

Shiueng-Bien Yang is the corresponding author of this manuscript titled " Constrained-Storage Variable-Branch Neural Tree for Classification". He declare that there is no conflict of interest in this manuscript.

REFERENCES

[1] Gelfand,S. B., Ravishankar,C. S., and Delp, E. J. (1991). an Iterative Growing and Pruning Algorithm for Classification Tree Design. IEEE Trans. Pattern Analysis and Machine Intell., 13(2), 163-174.

[2] Yildiz, O. T. and Alpaydin, E. (2001). Omnivariate Decision Trees. IEEE Trans. Neural Network, 12(6), 1539-1546.

[3] Zhao, H. and S. Ram (2004). Constrained Cascade Generalization of Decision Trees. IEEE Trans. Knowledgement and data Engineering, 16(6), 727-739.

[4] Gonzalo, M. M. and Alberto, S. (2004). Using All Data to Generate Decision Tree Ensembles. IEEE Trans. Systems, Man, Cybernetics, C, Applications and Reviews, 34(4), 393-397.

[5] Witold, P. and Zenon, A. S. (2005). C-fuzzy Decision Trees. IEEE Trans. Systems, Man, Cybernetics, C, Applications and Reviews, 35(4), 498-511.

[6] Wang, X. B. Chen, G. Q., and Ye, F. (2000). On the Optimization of Fuzzy Decision Trees. Fuzzy Sets and Systems, 112(3), 117-125.

[7] Deffuant, G., Neural units recruitment algorithm for generation of decision trees., Proceedings of the international joint conference on neural networks, 1 (1990) 637–642.

[8] Lippmann, R., An introduction to computing with neural nets., IEEE Acoustics, Speech, and Signal Processing Magazine. 4(2) (1987) 4–22.

[9] Sankar, A., & Mammone, R., Neural tree networks. In Neural network: theory and application, San Diego, CA, USA: Academic Press Professional, Inc. 1992, pp. 281–302.

[10] Sethi, I. K. and Yoo, J., Structure-driven induction of decision tree classifiers through neural learning, Pattern Recognition. 30(11) (1997) 1893–1904.

[11] Sirat, J., & Nadal, J., Neural trees: a new tool for classification, Neural Network. 1 (1990) 423–448.

[12] T. Li, Y. Y. Tang, and F. Y. Fang, A structure-parameter-adaptive (SPA) neural tree for the recognition of large character set, Pattern Recognit. 28(3) (1995) 315–329.

[13] M. Zhang and J. Fulcher, Face recognition using artificial neural networks group-based adaptive tolerance (GAT) trees, IEEE Trans. Neural Networks. 7 (1996) 555–567.

[14] G. L. Foresti and G. G. Pieroni, Exploiting neural trees in range image understanding, Pattern Recognit. Lett. 19(9) (1998) 869–878.

[15] H. H. Song and S.W. Lee, Aself-organizing neural tree for large set pattern classification, IEEE

Trans. Neural Networks. 9 (1998) 369–380.

[16] G. L. Foresti, Outdoor scene classification by a neural tree based approach, Pattern Anal. Applic. 2 (1999) 129–142.

[17] H. Guo and S. B. Gelfand, Classification trees with neural networks feature extraction, IEEE Trans. Neural Networks. (1992) 923–933.

[18] G. L. Foresti, An adaptive high-order neural tree for pattern recognition, IEEE Trans. Systems, man, cybernetics-part B: cybernetics. 34 (2004) 988-996.

[19] G. L. Giles and T. Maxwell, Learning, invariance, and variable-branchization in high-order neural networks, 26 (1987) 4972–4978.

[20] P. Maji, Efficient design of neural network tree using a single spilitting criterion, Nerocomputing. 71 (2008) 787–800.

[21] P. E. Utgoff, Perceptron tree: a case study in hybrid concept representation, Proc. VII Nat. Conf. Artificial Intelligence. (1998) 601–605.

[22] J. A. Sirat and J. P. Nadal, Neural tree: a new tool for classification, Network. 1 (1990) 423–438.

[23] G. L. Foresti and C. Micheloni, Generalized neural trees for pattern classification, IEEE Trans. Neural Networks. 13 (2002) 1540–1547.

[24] C. Micheloni, A. Rani, S. Kumarb, G. L. Foresti, A balanced neural tree for pattern classification, Neural Networks. 27 (2012) 81-90.

[25] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[26] J. Koza, *Genetic Programming*. Cambridge, MA: MIT Press, 1992.

[27] S. Grossberg, Ed., *Neural Networks and Natural Intelligence*. Cambridge, MA: MIT Press, 1988.

[28] D. Rumelhart and J. McClelland, Eds., *Parallel Distributed Processing: Explorations in Microstructure of Cognition*. Cambridge, MA: MIT Press, 1986.

[29] J. M. Zurada, Ed., *Introduction to Neural Systems*. St. Paul, MN:West, 1992.

[30] P. J. Angeline, G. M. Saunders, and J. B. Pollack, “An evolutionary algorithm that constructs

recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 5, no. 1, pp. 54–64, Jan. 1994.

[31] Mahmoudabadi H., Izadi M., Menhaj M.B. A hybrid method for grade estimation using genetic algorithm and neural networks. Computational Geosciences. 2009;13:91–101.

[32] Samanta B., Bandopadhyay S., Ganguli R. Data segmentation and genetic algorithms for sparse data division in Nome placer gold grade estimation using neural network and geostatistics. Mining Exploration Geology. 2004;11(1–4):69–76.

[33] Chatterjee S., Bandopadhyay S., Machuca D. Ore grade prediction using a genetic algorithm and clustering based ensemble neural network model. Mathematical Geosciences. 2010;42(3):309–326.

[34] Tahmasebi P., Hezarkhani A. IAMG09. Stanford University; California: 2009. (Application of Optimized Neural Network by Genetic Algorithm).

[35] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In International Joint Conference on Neural Networks, 2011.

[36] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Computer Science Department, University of Toronto, 2009.

[37] R. C. Gonzalez and R. E. Woods. Digital Image Processing. Addison–Wesley, Boston, MA, 1992.