Learning a Neural Tree

Article · October 1992

Source: CiteSeer

3 authors, including:

Andrew Jennings
RMIT University
30 PUBLICATIONS **427** CITATIONS
SEE PROFILE

Huan Liu
Arizona State University
457 PUBLICATIONS **22,447** CITATIONS
SEE PROFILE

Learning a Neural Tree

Wilson X. Wen Andrew Jennings Huan Liu

AI Systems, Telecom Research Laboratories
770 Blackburn Rd, Clayton, Victoria 3168, Australia

(in Proc. IJCNN'92, Beijing, China)

Abstract

A method to learn neural trees is proposed in this paper. Not only the weights of the network connections but also the structure of the whole network including the number of neurons and the interconnections among the neurons are all learned from the training set by our method. Issues about the optimization and pruning of the generated networks are investigated. Initial test results, comparisons with other learning methods and several possible applications are also discussed.

1 Introduction

It is well known that neural networks can learn connection weights from examples. However, is it possible to learn a neural network structure from examples? Structures of neural networks are usually designed by human experts. It is quite tricky to choose a good structure to fit for the learning task at hand. This makes the advantages of connectionist learning much less attractive. Furthermore, it turns out that the structure of a neural network is closely related to its final performance in some applications [Fukushima and Wake, 1991]. Therefore, a method to build a neural network automatically from the training data should be developed.

In this paper, a method to learn neural trees from training examples directly is proposed based on our early work [Wen *et al.*, 1992; Fang *et al.*, 1991]. The neural tree learned by this method is called SGNT (Self-Generating Neural Tree). This means that not only the weights but also the structure of the network, including the number of neurons, the connections between the neurons, and the weights of the neurons, are all learned from the training examples without any intervention from human experts.

2 Learning a Neural Tree

Like conventional Self-Organizing Neural Networks (SONN) [Kohonen, 1984], a Neural Tree (NT) is also a set of neurons each of which has a weight vector and some connections with other neurons. The difference is that conventional SONN's have a two dimensional grid structure while NT's have a tree structure. Therefore, NT's are good for hierarchical classification/clustering. Neural trees are much faster than conventional SONN on most conventional (sequential) computers since only the neurons along the search path need to be checked and updated for NT's while all the neurons in the grid have to be checked for conventional SONN. Before we describe the SGNT algorithms, some definitions related are given below:

Definition 1: An *instance* e_i is a real vector of attributes: $e_i = < a_{i1}, ..., a_{in} >$.

Definition 2: A *neuron* n_j is a ordered pair $< W_j, C_j >$, where W_j is the real weight vector of the neuron: $W_j = < w_{j1}, ..., w_{jn} >$, and C_j is the child neuron set of n_j.

Definition 3: An *SGNT* is a tree $< \{n_j\}, \{l_k\} >$ of neurons generated automatically from a set of training instances by the algorithm given below, where $\{n_j\}$ is the neuron/node set and $\{l_k\}$ is the link set of the tree. There is a directed link from neuron n_i to n_j, if and only if $n_j \in C_i$.

Definition 4: A neuron n_k in a neuron set $\{n_j\}$ is called a *winner* for an instance e_i if $\forall j, d(n_k, e_i) \leq d(n_j, e_i)$ where $d(n_j, e_i)$ is the distance between neuron n_j and instance e_i.

Any distance measure can be used, but we use a modified Euclidean distance measure as an example:

$$d(n_j, e_i) = \sqrt{\frac{\sum_{k=1}^{n} \rho_k \cdot (a_{jk} - a_{ik})^2}{n}}$$

where ρ_k is the weight for the k-th attribute. Note that, these weights are different from those weights for the SGNT neurons. The former are determined by a method based on information theory [Wen *et al.*, 1992] or a human expert prior to the learning phase while the latter are learned from the given training data set.

The SGNT algorithm is a hierarchical clustering algorithm. It is given in a pseudo-C language as follows:

Algorithm 1 (SGNT Generation/Training):

Input:

1. A set of training instances $E = \{e_i\}$, $i = 1, ..., N$.
2. A threshold $\xi \geq 0$.
3. A distance measure for each attribute/weight in instances/neurons.

Output: An SGNT generated from E.

Method:

```
copy(root,e0);
for(i=1,j=1;i<=N;i++) {
    minmumDistance = distance(ex,root);
    winner = oldWinner = root;
    minimumDistance = test(ei,root);
    if(minimumDistance>ξ) {
        if(leaf(winner)) {
            copy(nj,winner);
            connect(nj,winner);
            j++;
        }
        copy(nj,ei);
        connect(nj,winner);
        j++;
    }
    update(winner,ei);
}
```

where the routines are defined as follows:

1. `copy(n,e)`: create a neuron n and copy the attributes/weights in the instance/neuron e to n.
2. `distance(e,n)`: return the distance between instance e and neuron n.
3. `test(e,subRoot)`: find a winner in the current SGNT/sub-SGNT rooted by `subRoot` for instance e and return the distance between the winner and e.
4. `leaf(n)`: check a neuron n to see whether it is a leaf neuron in the current SGNT. A neuron in an SGNT is called a leaf neuron if it has no child neuron.
5. `connect`(n_0, n_1): connect neuron n_0 to n_1 making n_0 as a child neuron of n_1.
6. `update`(n_i, e_{k+1}): update the weight vector of neuron n_i by the attribute vector of e_{k+1} according to the updating rule (1) below.

In the NT generated by the above algorithm, each leaf neuron in the network corresponds to one or more training example and its weights are the averages of the corresponding attribute values of the example or examples. The weights of each non-leaf neuron n_j are the averages of the corresponding weights of all its children and therefore are also the averages of the corresponding attribute values of all the examples at the bottom of the subtree rooted by n_j (in this case, we say that the examples are covered by n_j). That is

$$w_{jk} = \frac{1}{N_j} \sum_{i=1}^{N_j} a_{i,k}$$

where w_{jk} is the kth weight of neuron n_j, N_j is the number of examples covered by n_j, and $a_{i,k}$ is the value of the $k$the attribute of i-th example covered by n_j. Thus we have the successive updating rule:

$$w_{jk,i+1} = w_{jk,i} + \frac{1}{i+1} \cdot (a_{i+1,k} - w_{jk,i}). \tag{1}$$

where $w_{jk,i}$ is the k-th weight of n_j after we have seen the first i examples which are covered by n_j. It is easy to see that this is exactly the standard SONN updating rule when the gain term is $\frac{1}{i+1}$. The difference is that, here, the updating produces exactly the average of the weights of all the examples covered by the neuron.

Example 1. Suppose we have 4 one-dimensional training instances $e_1 = 1$, $e_2 = 2$, $e_3 = 3$, and $e_4 = 4$, and $\xi = 0$. The SGNT generation is as follows:

1. e_1 is just copied to a neuron n_0 as the root, $w_{00} = 1$.
2. w_{00} is updated by e_2 to $1 + \frac{1}{2}(2-1) = 1.5$, n_1 and n_2 are generated as the children of n_0 (Fig. 1 a)[1], and $w_{10} = 1$, $w_{20} = 2$.
3. w_{00} is updated by e_3 to $1.5 + \frac{1}{3}(3-1.5) = 2$. The winner in $\{n_0, n_1, n_2\}$ is n_2 since $d(e_3, n_0) = 1.5$ (before w_{00} is updated), $d(e_3, n_1) = 2$, and $d(e_3, n_2) = 1$; and thus, w_{20} is updated to $2 + \frac{1}{2}(3-2) = 2.5$. n_3 and n_4 are generated as the children of n_2 and $w_{30} = 2$, $w_{40} = 3$ (Fig. 1 b).
4. e_4 updates w_{00} to $2 + \frac{1}{4}(4-2) = 2.5$, w_{20} to $2.5 + \frac{1}{3}(4-2.5) = 3$, w_{40} to $3 + \frac{1}{2}(4-3) = 3.5$, and n_5 and n_6 are generated as the children of n_4 with $w_{50} = 3$ and $w_{60} = 4$ (Fig. 1 c).

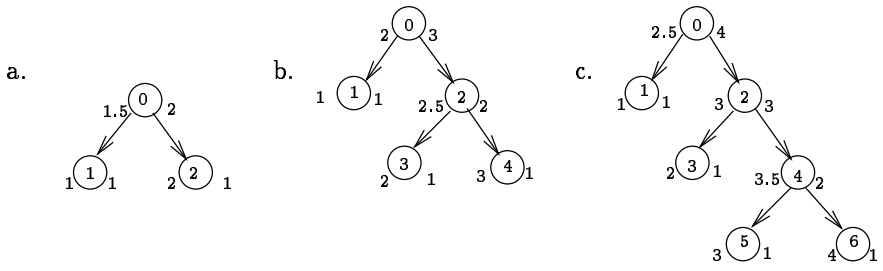

Figure 1: Network generation in Example 1.

After a network is built, it can be used to classify test instances (Algorithm 2). In Fig. 1 c, suppose we have a test instance 3.1, n_2 is found to be the winner in $\{n_0, n_1, n_2\}$. Then, in $\{n_2, n_3, n_4\}$, n_2 is again found to be the winner. The search thus stops and the global wining class is the one represented by n_2.

3 Optimization and Pruning

It is easy to see from Example 1 that the NT in Fig. 1 c is not optimal. For example,

[1] Where, the number inside the circle is the neuron number, the number on the left is the weight, and the number on the right is the number of examples covered by the neuron

1. The network is not in the best shape, eg. $d(n_3, n_2) = 1 > d(n_3, n_0) = 0.5$. This is obviously not desirable since our purpose is to make neurons clustered according to their distance.

2. Suppose we have a testing instance 2. The winner in $\{n_0, n_1, n_2\}$ is n_0, and the search stops because n_0 is the root. It is easy to see that neuron n_3 should be the true winner which can never be reached.

It seems that we need some optimization to make the network in a better shape and more accurate for clustering/classification.

To make the network in better shape, we adopt Mckusick and Langley's method [1991]:

Definition 5. A neuron n is *Horizontally Well Placed* (HWP) in an SGNT, if $\forall n_s \in C_p, d(n, n_p) \leq d(n, n_s)$, where n_p is the parent of n.

Definition 6. n is *Vertically Well Placed* (VWP), if $d(n, n_p) \leq d(n, n_g)$, where n_g is the parent of n_p.

Definition 7. An NT is *well organized* (WO) if all of its neurons are both HWP and VWP.

After an NT is generated we may need check it to see whether it is WO. If not an optimization may need to be performed to make it in a better shape.

If n is not HWP, n should be moved down by one layer under n_s when $d(n, n_p) > d(n, n_s)$. In this case, n_s should be copied to its another new child and the weights of n_s should be updated to

$$w_{sj} = \frac{N_s \cdot w_{sj} + N_n \cdot w_{nj}}{N_s + N_n}.$$

where N_s is the number of examples covered by n_s, etc. We should also increase N_s to $N_s + N_n$.

If n is not VWP, n should be moved up by one layer as a child of n_g. The weights of n_p are updated to

$$w_{pj} = \frac{N_p \cdot w_{pj} - N_n \cdot w_{nj}}{N_p - N_n},$$

and we also need to reduce N_p to $N_p - N_n$.

Example 1 (continue). In Fig. 1 c, n_3 and n_5 are not VWP. Move n_3 up to make it as a child of n_0 and decrease the weight of n_2 to $\frac{3\times3-1\times2}{3-1} = 3.5$ and the number of examples covered by n_2 is $3 - 1 = 2$. At this stage, because the updated n_2 and n_4 are identical, n_4 is merged with n_2, and the network is shown in Fig. 2 a. However, the network is still not well organized because n_1 is not HWP after we moved n_3 up. Therefore, we move n_1 down one layer and finally obtain a well organized network (Fig. 2 b). Note that all of the training examples can be perfectly retrieved from this neural network.

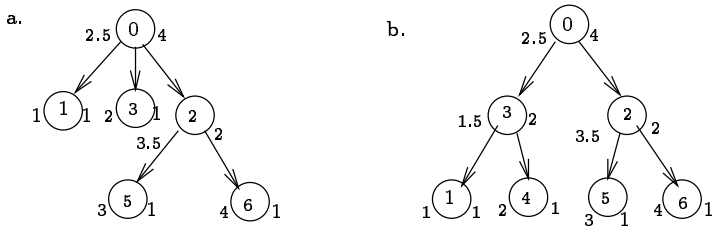

Figure 2: Optimization in Example 1

Another way to get around the problems described above is to train the network repeatedly and prune away the dead branches which do not grow during the repeated training. A branch is called *dead* during repeated training if the number of the examples covered by the root of the branch does not increase. In Fig. 1 c, repeat the training one epoch, we have a network shown in Fig. 3 a. The numbers of examples covered by all neurons grow except n_3 and n_5. Thus, we prune n_3 and n_5 away and obtain the neural network in Fig. 3 b. If we keep repeatedly training, the structure will keep unchanged, ie. it reaches a stable state.

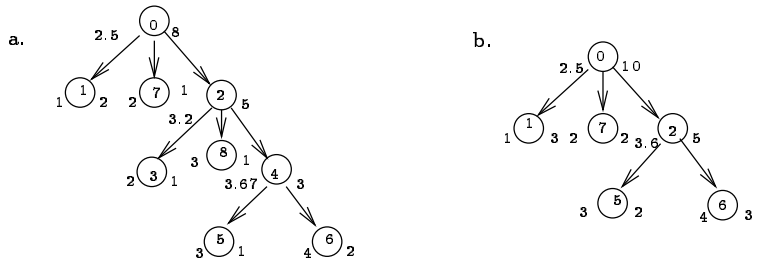

Figure 3: Pruning in Example 1

The network generated for some application may have an unacceptable big size in numbers of neurons and/or connections. For example, in a public domain data set of mushroom classification [Schlimmer, 1987] there are 8124 examples each with 23 attributes. Randomly choosing 2000 of them as training set we generate a network with 3109 neurons in 9 layers. This is too big for practical use

To reduce the size of the SGNT, we adopt a method similar to Quinlan's method [1987] used to simplify decision trees. The SGNT is tested by the training set. The number of errors occurred in the test is accumulated and recorded at each neuron. The subnetwork rooted by neuron n_j which satisfies the following conditions may be pruned away (but n_j itself is still kept):

$$E + \frac{1}{1+\beta} \leq \gamma \cdot \sigma(\gamma)$$

where E and E' are the numbers of errors occurred at n_j in the pruned and unpruned network, respectively, $\sigma(x)$ is the standard error of x, β is the number of children which n_j has, and

$$\gamma = E' + \frac{\beta}{1+\beta}$$

For the mushroom data set, after simplification, only 149 neurons left, the simplified network has the same accuracy in classification for both training and test sets as before simplification.

4 COMPARISONS AND CONCLUSIONS

Two SGNT systems have been developed on SUN SPARC 1 and IBM-PC/AT with X-Window and Windows 3 interfaces, respectively. Some experiments have been conducted to evaluate the system and the method. The results for unsupervised learning seem quite encouraging:

Data Set	Number of Attributes	No. examples		Test results		The best results of other methods
		Training	Testing	Training	Testing	
US Voting	17	300	135	100%	95.1%	95%
Mushrooms	23	2000	6124	100%	99.9%	94% for only 3000 instances
Monk 1	7	124	435	94.0%	82.6%	82.7%
Monk 2	7	169	435	92.0%	78.2%	71.3%
Monk 3	7	122	435	98.1%	84.5%	68.0%

Another important advantage of our method is that it is much faster than most of the popular unsupervised learning methods such as COBWEB, CLASSWEB, and conventional SONN's. For example, it takes 924.63, 1212.26, and 884.85 seconds in average to solve the MONK's problems [Thrun and others, 1991] by CLASSWEB while it takes only 1.47, 2.37, and 2.69 seconds by SGNT.

It turns out that the new method has quite a few attractive advantages over the conventional SONN and the other unsupervised learning methods. The network design effort is significantly reduced. All we need to build a network is to choose a set of typical training examples for the applications at hand. The tree

structure generated is a good fit for the application. Therefore, good generalization capability and results for clustering/concept formation can be obtained. There is not much waste of neurons and connections to solve our task. Thus, large application problems such as mushroom classification and image compression can be solved without implementation difficulties. The method might well be a very natural way to combine the neural networks and symbolic AI because of its good concept formation capabilities or ability for generating explanation automatically and robustness against noises. The possible application areas of SGNT include prediction/diagnostic systems, image coding, and information retrieval systems.

ACKNOWLEDGEMENT: The Permission of the Director, TRL to publish this paper is acknowledged.

References

[Fang *et al.*, 1991] L. Fang, A.Jennings, W.Wen, K.Li, and T.Li. Unsupervised learning in a self-organizing tree. In *Proc. IJCNN'91 (International Joint Conf. on Neural Networks)*, Singapore, Nov. 1991.

[Fukushima and Wake, 1991] K. Fukushima and N. Wake. Handwritten alphanumeric character recognition by the neocognition. *IEEE Trans. on Neural Networks*, 2, No. 3:355–365, 1991.

[Kohonen, 1984] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin, 1984.

[McKusick and Langley, 1991] K. McKusick and P. Langley. Constrains on tree structure in concept formation. In *Proc. IJCAI'91*, volume 2, pages 810–816, Sydney, Aug. 1991. Morgan Kaufmann.

[Quinlan, 1987] J.R. Quinlan. Simplifying decision trees. *Int.J.Man-Machine Studies*, 27:221–234, 1987.

[Schlimmer, 1987] J.S. Schlimmer. Concept acquisition through representational adjustment. Technical report 87-19, Department of Information and Computer Science, University of California, Irvine, 1987.

[Thrun and others, 1991] S.B. Thrun et al. The MONK's problems: A performance comparison of different learning algorithms. Tech Report CMU-CS-91-197, Carnegie Mellon University, Dec. 1991.

[Wen *et al.*, 1992] Wislon X. Wen, Huan Liu, and Andrew Jennings. Self-generating neural networks. In *Proc. IJCNN'92 (International Joint Conf. on Neural Networks)*, Baltimore, June 1992.