S2.25

NEURAL TREES – USING NEURAL NETS IN A TREE CLASSIFIER STRUCTURE

Jan-Erik Strömberg, Jalel Zrida[1] and Alf Isaksson
Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden

Abstract

In this paper, we are combining the concept of tree classifiers with the popular neural net structure. Instead of having one large neural net to capture all the regions in the feature space, we suggest the compromise to use small single output nets at each tree node. This hybrid classifier is referred to as a *neural tree*. The performance of this classifier is evaluated on real data from a problem in speech recognition. When verified on this particular problem, it turns out that the classifier concept drastically reduces the computational complexity compared with conventional multilevel neural nets. We also notice, that this idea makes it possible to grow trees on-line, from a continuous data stream.

1 Introduction

In this paper, we are combining the concept of tree classifiers, presented in [2], with the popular neural net structure. The basic idea behind tree classifiers, is to organize the class labeling into a tree search, which is guided by properly chosen questions based on the feature vector. These node questions, or splits, are represented as internal nodes of the tree and the class labels, as leaves.

The most popular kind of node question is a simple scalar threshold test on one component of the feature vector. Unfortunately, this kind of simple node question normally leads to complex tree structures when capturing non-trivial regions in the feature space. The more complex questions asked, the simpler tree structure we get. The main point in this paper is the observation that conventional neural nets allows us to ask arbitrarily complex questions. Instead of having one large neural net to capture all the regions in the feature space, we suggest the compromise to use small single output nets at each tree node. This hybrid classifier is what we refer to as the *neural tree*.

The neural tree classifier is tested on the problem presented in [5], namely to classify vowels in continuous speech. The same data base is used as in [1] and [5].

2 Tree classification

All the classification methods used in this paper are based on the principle of *induction* as introduced by Breiman and co-workers in [2]. Related work by Quinlan and Cockett is presented in [8] and [3].

[1]Now with King Fahd University of Petroleum and Minerals, Dharan, Saudi Arabia

The induction process can be best described with no loss of generality by the following two–class problem:

Suppose that in a given node of the classifying tree we have a set S of objects consisting of p objects of class P and n objects of class N. We then define an *impurity measure* $H(S) \geq 0$ as a function with the following properties:

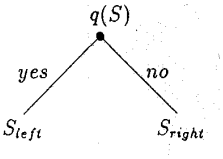
1. $H(S)$ maximum if $p = n$
2. $H(S) = 0$, if $p = 0$ or $n = 0$

One popular choice of function fulfilling these requirements is the Gini index (see [2]):

$$H(S) = \frac{2np}{(p+n)^2}. \tag{1}$$

Now, if a decision or question q is used at this node, it will partition S into two mutually exclusive sets S_{left} and S_{right}. S_{left} then represents the set of objects in S which are split to the left by the decision q, and S_{right} the objects split to the right. If the set S_{left} contains p_l objects of class P and n_l objects of class N, and S_{right} contains p_r objects of class P and n_r objects of class N, then the total purity gained by using decision q at this node is

$$\Delta H(S,q) = H(S) - \frac{p_l + n_l}{p+n} H(S_{left}) + \frac{p_r + n_r}{p+n} H(S_{right}).$$



The selection criterion or splitting rule selects, at each step of the induction process, the attribute which provides the most purity gain. The process stops when there is no more significant purity to gain, *i.e.*, all the terminal nodes have only dominantly one–class objects. This search scheme is of course sub-optimal and will, in general, not produce the optimal tree.

There are mainly two ways of generalizing the above to the multi-class problem, with a set C of classes. The first approach is to search for two super-classes C_1 and C_2 such that $C_1 \cup C_2 = C$ and $C_1 \cap C_2 = \emptyset$. This conglomeration of the original class labels into two separate super-classes is done at each node of the tree and the selection of best split may once again be guided by (1). The second approach is to generalize (1) to a multi class case. This brings us to

$$H(S) = \sum_{i \neq j} p(i|S)p(j|S),$$

CH2977-7/91/0000-0137 $1.00 © 1991 IEEE

where $p(i|S)$ and $p(j|S)$ are the probabilities that an arbitrary object, taken from the set of objects S, belongs to class i or class j, respectively. It is difficult to say something general about what method to prefer, but according to [2] the first method tends to give a more unbalanced tree than the second in most of the observed cases.

The simplest kind of node question is to test one single attribute at each node, *i.e.*, to look for hyperplanes which are orthogonal to the axes of the attribute-spanned space. A question or decision at a node k of the tree then has the following form:

$$x_j(t) \leq \tau_k,$$

with τ_k being a threshold value for the attribute x_j (t represents time or sample index). These hyperplanes can be found by considering all the possible cuts along the axis for each attribute, computing the purity gain, and selecting as best attribute/cut the one which gains the most purity.

Pruning

Realistic training sets are naturally corrupted by non-systematic errors on either the attribute values or the class assignment. This *noise* is almost always present and significantly affects the results of the training since we do not know if we are in fact just modeling the sample unique noise. This problem is also referred to as *over training*. There are basically two ways of controlling this problem in the tree growing process. The first and most obvious approach, is to find some intelligent stop growing criterion based on the impurity measure (1). In [2] it is noted that this approach is in general not successful, since the choice of criterion turns out to be a very difficult one in general. Another and completely different approach is therefore suggested by Breiman and co-workers: *minimum cost-complexity pruning*.

This pruning is based on defining, for an induction tree, a parameterized cost–complexity measure which is a linear combination of the misclassification cost of the induction tree and its *complexity*. The complexity chosen is defined as number of leaves and the pruning algorithm produces a nested sequence of pruned subtrees (collapsing to the root node). This pruning is usually based on the training set, i.e. the same set as used for growing the tree, but the selection of best pruned tree is preferably made according to a separate validation data set.

3 Neural trees

The main limitation with the induction algorithm presented above is the type of questions used. Checking one attribute at the time often results in a large tree. A natural extension is of course to include more complex questions at the nodes, which is also discussed in [2]. As a solution to this problem we suggest to train small neural nets at the nodes to find the questions. This is what we refer to as the *neural tree*.

A *neural network* or *multi-level perceptron* (MLP) is a parameterized nonlinear mapping $y(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$. The computational structure is composed of several interconnected nonlinear processor elements. These are called *perceptrons* and are pre-arranged in a pile of layers. In addition to the input and output layer, a neural net can have an optional number of hidden layers.

The interconnections between nodes are defined by a set of weight parameters. A given node is only connected to the nodes in the next layer. There are no interconnections within a layer and no interconnections "backwards" between layers.

The main functional component of the perceptron, is the non-linear function f, traditionally taken to be the *sigmoid* function

$$f(x) = \frac{1}{1+e^{-x}}.$$

The processing of data at the perceptron level is done as follows: Given a node j in a given layer k of a neural net its state is given by

$$y_j^k = f(\sum_i w_{ij}^k x_i^k),$$

with w_{ij}^k being the weight parameter connecting perceptron j at layer k with input x_i^k.

Basically, the neural tree training works as follows. For each node of the tree, take the training data set at that node and train a neural net as described below. Here x represent the attribute values and $d(x)$ the corresponding true class according to

$$d_j(x) = \begin{cases} 1 & \text{if } j = c(x) \\ 0 & \text{otherwise} \end{cases}$$

where $c(x) \in \{1, 2, ..., N_c\}$ is the class label corresponding to x. Therefore we need a neural net with $n = N_a + 1$ inputs and $m = N_c$ outputs, N_c being the number of classes and N_a the number of attributes. The extra input is set to a constant value (typically 1), to enable a constant offset in the mapping from $\mathbb{R}^{N_a}$ to the set of classes.

When the neural net has been trained we expect that a sample from class c produces $y_c = 1$ and the remaining outputs equal to zero. Therefore a natural choice of node question is "$y_j < 0.5$?", which in the ideal case singles out one unique class, namely j, to be routed in the "No"-direction of the tree. Since this gives us N_c potential splits, we choose the one which gains the most purity when splitting the training data. The weight parameters corresponding to this particular output node, will thus define the selected question to ask at that node.

Like in the Breiman method, the tree growing continues until there is no more significant purity to gain or the training data set is too small, in which cases the process stops. The current node is then turned to a leaf, which is labeled by the majority class.

Neural net training

The most popular adjusting or learning method used today is the *back propagation* algorithm, see [6]. This algorithm uses gradient updates to minimize the loss function

$$V = \frac{1}{N}\sum_{t=1}^{N} \varepsilon^T(t)\varepsilon(t); \quad \varepsilon(t) = d(x(t)) - y(t).$$

If we take a single layer neural net for simplicity, the minimization may be carried out through an iterative search procedure, where at each iteration, the weight vector w_j is updated according to

$$w_j^{(l+1)} = w_j^{(l)} + \eta\delta_j.$$

The constant η is a gain term which is appropriately chosen to speed the convergence and insure the stability of the algorithm, while δ_j represent a search direction based on the

loss function V. The back propagation algorithm, being a gradient method, therefore uses

$$\delta_j = -\frac{\partial V}{\partial w_j} = -\frac{2}{N}\sum_{t=1}^{N} y_j(t)(1-y_j(t))\varepsilon_j(t)x(t). \quad (2)$$

In neural trees however, we are only using small size neural nets at the nodes. Therefore, in the test case to follow we have chosen to use a Gauss-Newton minimization scheme instead. This scheme is computationally heavy when the number N_w of weight parameters is large, since we need to invert an $(N_w \times N_w)$ matrix. Gauss-Newton, however, results in faster convergence than the back propagation algorithm. The Newton direction is given by

$$\delta_j = -\left(\frac{\partial^2 V}{\partial w_j^2}\right)^{-1}\frac{\partial V}{\partial w_j}, \quad (3)$$

and the Gauss-Newton approximation by

$$\frac{\partial^2 V}{\partial w_j^2} \approx \frac{1}{N}\sum_{t=1}^{N}\psi_j(t)\psi_j^T(t)$$

where

$$\psi_j(t) = -\frac{d\varepsilon_j(x(t))}{dw_j} = y_j(t)(1-y_j(t))x(t).$$

Comparing (2) and (3) we find that the only difference is that in the Newton direction we have compensated the gradient direction with the Hessian. A thorough treatment of this topic can be found in [4].

4 Test Case

One of the most interesting and hard problems in the speech recognition area is to extract phonemes from natural speech. This is the problem chosen as test case for the neural tree concept.

4.1 The Göteborg Data Set

The physical data set used in the test was created by P Knagenhjelm and P Brauer at Chalmers University of Technology and was originally used in the paper [1]. The data set is based on a recording of a male professional Swedish speaker reading a sample text (6 minutes long). The speech is sampled at 8 kHz and pre-filtered. From this, 16 reflection coefficients are identified. Given the reflection coefficients, 16 cepstrum coefficients are calculated, see [7].

After segmentation, each vowel or time slot (16 ms duration) is manually assigned to a class label, corresponding to 12 different phonemic classes. This way the data set consists of 3440 samples, each having 16 attribute values (the cepstrum coefficients) and a unique class label (an integer in $\{1,2,\ldots,12\}$). This data set is furthermore divided into two (approximately) equally distributed sets: the learning and validation data sets respectively. The class distribution of the learning set is presented in the following table.

Label	Vowel	Entries	Label	Vowel	Entries
1	*y*	38	7	*E*	26
2	*i*	197	8	*U*	62
3	*a*	591	9	*ä*	147
4	*A*	382	10	*Ö*	26
5	*e*	121	11	*u*	35
6	*o*	53	12	*ö*	42

4.2 The Breiman Induction

The Breiman induction algorithm applied to the Göteborg data set results in a tree with 235 leaves. Estimating the classification rate via the learning and validating sets, we get 86.7% and 74.7% respectively. Pruning the tree using the learning set, we have selected the minimum misclassifying subtree according to the validation set. The characteristics of the pruned tree, as well as the original tree, are summarized in the table below.

	# of Leaves	Class Rate on Learning Set	Class Rate on Validation Set
Before Pruning	235	86.7%	74.7%
After Pruning	93	84.8%	74.9%

4.3 The Single Layer Neural Tree

The pruned Breiman tree has a total of 92 internal nodes or node questions. This clearly indicates that orthogonal hyperplanes are fairly inappropriate for this problem. The natural "next step" is to check a single layer neural tree, *i.e.* a tree where the node questions are based on linear combinations of the attributes. We summarize the performance of the resulting neural trees in a table.

	# of Leaves	Class Rate on Learning Set	Class Rate on Validation Set
Before Pruning	45	96.9%	80.3%
After Pruning	19	91.5%	83.3%

Apart from a significant improvement in classification rate, we also notice a drastic reduction in complexity. In fact, we can hardly expect any further improvements when using more complex questions, since the number of leaves is approximately equal to the number of classes.

A more detailed specification of the classifier's performance is in terms of its confusion matrix. Element (i,j) of this matrix, is the estimated probability (in percent) that a class i object is classified as j. The pruned tree, has the following confusion matrix:

$$\begin{bmatrix}
58 & 24 & 0 & 0 & 0 & 0 & 3 & 0 & 8 & 0 & 5 & 3 \\
3 & 83 & 2 & 0 & 9 & 0 & 1 & 1 & 2 & 0 & 0 & 0 \\
0 & 0 & 93 & 4 & 0 & 1 & 0 & 2 & 0 & 1 & 0 & 0 \\
0 & 1 & 5 & 85 & 0 & 0 & 1 & 1 & 8 & 0 & 0 & 0 \\
0 & 10 & 1 & 1 & 80 & 0 & 1 & 0 & 7 & 0 & 0 & 0 \\
0 & 0 & 21 & 2 & 0 & 68 & 2 & 6 & 0 & 0 & 0 & 2 \\
4 & 12 & 4 & 4 & 0 & 0 & 54 & 4 & 4 & 8 & 4 & 4 \\
0 & 0 & 16 & 3 & 0 & 2 & 0 & 61 & 8 & 6 & 2 & 2 \\
0 & 1 & 2 & 11 & 2 & 0 & 1 & 0 & 82 & 0 & 0 & 0 \\
0 & 4 & 50 & 4 & 0 & 0 & 0 & 8 & 0 & 31 & 4 & 0 \\
9 & 3 & 0 & 3 & 0 & 0 & 3 & 6 & 3 & 0 & 74 & 0 \\
2 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 2 & 2 & 5 & 83
\end{bmatrix}$$

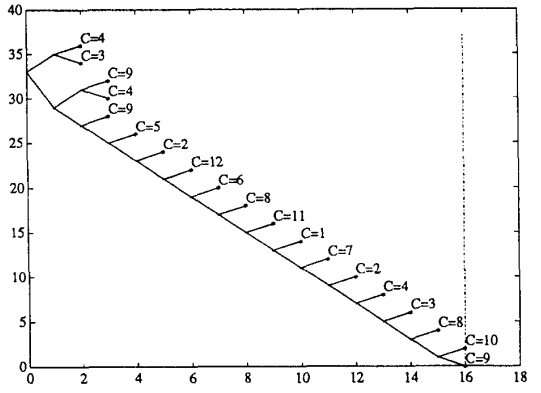


Figure 1: The pruned single layer neural tree.

Since the tree is so small, we may even plot it in its entirety, see Figure 1. Here we notice something quite expected, namely that the neural tree becomes highly unbalanced. As mentioned before, this is mainly explained by the neural net training, where we want each output of the net to single out one class from the others. However, it could be argued, that this in combination with the split selection criterion (1), produces a tree that singles out highly frequent classes near the root. This is of course true only if the learning data set is well enough representing the underlying process.

4.4 Conventional MLP vs Tree Classifiers

To relate the performance of the tree classifiers with other approaches, we consider the best classifier from [5], which was a conventional MLP. The MLP used had 40 nodes at the first hidden layer, 20 nodes at the second and 12 nodes at the output layer. The results obtained are summarized below where we also repeat the best results from the trees used above.

Type of Classifier	Class Rate on Validation Set	Computational Complexity
MLP network	83.7%	1520
Single Layer NT	83.3%	256
Breiman Tree	74.9%	-

First we notice that the performance of the single layer neural tree and the MLP network are very close with respect to classification rate. To give a crude measure of the computational complexity we have in the table also presented the number of multiplications needed to classify one object. Observe, that the complexity of the sigmoid has been skipped completely. Also notice that the complexity obtained for the tree is a worst case measure. We could expect the tree to behave much better on the average.

5 Conclusions

The obtained classification rate for the neural tree is comparable with the results presented in [5], but neural trees have the following additional advantages:

- Lower computational complexity when compared with conventional neural nets.
- Lower structural complexity compared with the Breiman tree (simple scalar threshold).
- Neural trees may very well be implemented as a parallel computer in VLSI (implementing a conventional MLP in VLSI is in general hard because of the immense amount of connections).
- The complexity of the classifier can be trimmed to any level via a performance guided pruning scheme.
- On-line building of tree classifiers possible. This has been implemented using back propagation training, and so far the results are most promising.

Finally we also make the following remarks:

- The Gauss-Newton method behaves considerably better than the established back propagation neural net training - at least when used on small size nets as in neural trees.
- In order to reduce the worst case complexity of the neural tree, we naturally need some way to balance it. There are some ideas how this could be done, but so far nothing has been implemented or tested in our work.

References

[1] P. Brauer and P. Knagenhjelm. Infrastructure in Kohonen maps. In *Proc. of IEEE international conf. on acoustics, speech and signal processing*, volume 1, pages 647–650. IEEE, 1989.

[2] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and regression trees.* The Wadsworth Statistics-Probability Series. Wadsworth & Brooks, Monterey, California, 1984.

[3] J.R.B. Cockett, J. Zrida, and J.D. Birdwell. Stochastic decision theory. *Probability in the Engineering and Information Sciences*, 3:13–54, 1989.

[4] J.E. Dennis, Jr. and R.B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[5] P. Knagenhjelm and P. Brauer. Classification of vowels in continous speech using MLP and hybrid net. *Neurospeech*, Special Issue, 1990.

[6] Richard P. Lippmann. An introduction to computing with neural nets. *IEEE Transactions on Automatic Control*, April:4–22, 1990.

[7] J.D. Markel and A.H. Gray, Jr. *Linear prediction of speech.* Springer-Verlag, Berlin, 1982.

[8] J.R. Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.