Ntree: Tools for Neural Trees

Alois P. Heinz* Christoph Hense*

Abstract

Ntree is a C library of tools for the construction, training, and evaluation of Neural Trees. It contains an improved version of the CART algorithm for the construction of binary decision trees that incorporate splits with respect to fuzzy sets. These trees can be regarded as (or translated to) Neural Trees. For their training the Ntree library implements the AFLN algorithm. With these algorithms one is able to learn functions of the kind $f : D_1 \times \cdots \times D_n \to I\!R$ from a given set of training data where for each $i \in \{1, ..., n\}$ D_i may either be $I\!R$ or an unordered set of symbolic attribute values. This document mainly describes the usage of the Ntree library.

*Institut für Informatik, Universität Freiburg, Am Flughafen, Geb. 51, D-79110 Freiburg, Germany. E-mail: heinz@informatik.uni-freiburg.de

Contents

1 Introduction

The Ntree software was developed during our research on a special kind of artificial neural network called Lazy Neural Tree or Adaptive Fuzzy Logic Network (AFLN). The software as well as this document is available under the address:

`ftp://ftp.informatik.uni-freiburg.de/pub/neural/ntree.tar.gz`

By copying the software you agree to the Ntree License which gives you only the legal permission to copy the software and distribute unmodified copies or modify it for you own purpose. For our protection we want to make certain that everyone understands that there is no warranty of any kind for the Ntree software.

1.1 The AFLN algorithm

We presume that the reader is familiar with decision trees (classification trees, regression trees) as described in [1]. We will now briefly outline the basic idea of the AFLN algorithm. For a more detailed coverage see [3, 4, 5, 6].

Let S be a training set of evaluated attribute vectors for a n-dimensional regression problem with real valued attributes. Let T be a binary regression tree with axis parallel split questions which was obtained from S by the CART algorithm. Let T also denote the corresponding function $T : I\!R^n \rightarrow I\!R$. T is piecewise constant on axis parallel hyper-rectangles in $I\!R^n$.

A recursive evaluation procedure in pseudo C for T can be stated as follows. Each inner *node* of T has the fields *node.dimension*, *node.split-value*, *node.left-child*, and *node.right-child*. Each leaf has the field *node.evaluation*. In order to evaluate T for a given attribute vector *vector* function *evaluate()* is called for the root node.

```
evaluate(node,vector)
{
if { node is leaf }
     return node.evaluation;
else if ( vector[node.dimension] <= node.split-value )
     return evaluate(node.left-child,vector)
else
     return evaluate(node.right-child,vector)
}
```

In order to smooth the function represented by T the hard split questions of the inner nodes are replaced by fuzzy split questions. Let $f : I\!R \rightarrow I\!R$ be the following transition function

$$f(x) := \begin{cases} 0 & \text{if } x \leq -1, \\ \frac{1}{2} + x\left(1 - \frac{|x|}{2}\right) & \text{if } -1 < x < 1, \text{ and} \\ 1 & \text{if } 1 \leq x. \end{cases}$$

T is transformed into T^* by introducing a new real valued positive parameter *node.softness-radius* for each inner node of T. Now the new evaluation procedure for T^* can be given as follows:

```
evaluate*(node,vector)
{
if { node is leaf }
     return node.evaluation;
else if (vector[node.dimension] <= node.split-value - node.softness-radius)
     return evaluate*(node.left-child,vector);
else if (vector[node.dimension] >= node.split-value + node.softness-radius)
     return evaluate*(node.right-child,vector);
else /* the transition case */
     {
     l := f(  ( node.split-value - vector[node.dimension] )
            / ( node.softness-radius ) );
     r := 1 - l;
     return (    l * evaluate*(node.left-child,vector)
               + r * evaluate*(node.right-child,vector) );
     }
}
```

Now $T^* : I\!R^n \rightarrow I\!R$ is smooth, and if we choose the softness-radii small enough, is identical to T on S.

Since T^* is a smooth function the mean square error of T^* on S (or any new set of training vectors) is partially differentiable with respect to all free parameters, the split values, the softness radii, and the leaf evaluations. Thus T^* can be retrained on S by applying an appropriate gradient descent method.

T^* can be regarded as an artificial neural network with n attribute values as inputs and the evaluation as output connection. It can be easily translated into a 3-layer neural network. The CART algorithm provides the topology and initial values of the network and the AFLN learning algorithm optimizes the parameter settings.

1.2 Attributes of symbolic type

As an additional feature attributes of symbolic, non-ordered type are introduced. For each symbolic attribute the set of possible values must be finite and its size must be known.

An inner node of a regression tree with a split with respect to a symbolic attribute divides the set of possible attribute values into two subsets, one for the left successor of the node and the other for the right successor. Let A be a symbolic attribute of the feature space and $|A|$ its number of possible values. When our version of the CART algorithm looks for the best split of a node k with respect to A not all $2^{|A|-1} - 1$ possible splits of A are investigated but only $|A| - 1$ of them according to the following search strategy:

1. For each $a \in A$ calculate the average evaluation of the training vectors with attribute value a.

2. Sort A with respect to the average evaluation.

3. Scan through A using the new sort order and consider all splits which divide A into two connected subsets.

The complexity of applying this strategy is of the same order than the complexity of looking for the best split for an ordered attribute. Thus the complexity of CART is not increased by the introduction of symbolic attributes.

A regression tree T is modified by introducing a new field *node.set* to all inner nodes which have a split with respect to a symbolic attribute. This field contains an array $[s_1, \ldots, s_{|A|}]$ that encodes the division of A into subsets. For each $i \in \{1, \ldots, |A|\}$ $s_i = 1$ is set if the ith attribute value belongs to the left subset and $s_i = 0$ else. The evaluation procedure becomes:

```
evaluate(node,vector)
{
if { node is leaf }
     return node.evaluation;
else if { node splits with respect to an ordered attribute }
     {
     ...
     }
else  /* node splits with respect to a symbolic attribute */
     {
     if ( node.set[vector[node.dimension]] == 1 )
          return evaluate(node.left-child,vector);
     else
          return evaluate(node.right-child,vector);
     }
}
```

After transforming T to an AFLN T^* we allow the s_i, $i \in \{1, \ldots, |A|\}$, to carry any value from the real interval $[0, 1]$. Thus *node.set* can be regarded as a fuzzy set. The evaluation procedure becomes:

```
evaluate*(node,vector)
{
if { node is leaf }
     return node.evaluation;
else if { node splits with respect to an ordered attribute }
     {
     ...
     }
```

```
else /* node splits with respect to a symbolic attribute */
     {
     l := node.set[vector[node.dimension]];
     r := 1 - l;

     if ( l == 1 )
          return evaluate(node.left-child,vector);
     else if ( r == 1 )
          return evaluate(node.right-child,vector);
     else
          return (    l * evaluate(node.left-child,vector)
                   +  r * evaluate(node.right-child,vector) );
     }
}
```

The components of the *node.set* sets are regarded as free parameters and incorporated into the AFLN training process. Their initial values are either 0 or 1 and after training they take on values from the interval $[0, 1]$.

2 Writing applications with Ntree

Suppose you have two files of evaluated feature vectors: *training-data* and *test-data*. Then the following code builds a regression or classification tree from *training-data*, checks its performance with *test-data*, trains the corresponding AFLN with *training-data* and finally checks its performance with *test-data*. All domain specific parameters are given in the file *parameters:*

```
#include "ntree.h"
#include "ctree.h"
...
{
ntree_t *tree;
tset_t *tset;
parameter_t *parameters;
tset_t *lset;

/* load the parameters */
parameters = ntree_parameter_load("parameters");

/* load the training and the test set */
lset = tset_load("training-data",parameters);
tset = tset_load("test-data",parameters);

/* construct the classification or regression tree */
tree = ctree_construct(lset,parameters);
```

```
/* check its performance */
ntest_test(tree,tset,parameters);

/* save it */
ntree_store(tree,"CART");

/* train it as an AFLN */
ntree_train(tree,lset,tset,parameters);

/* check the performance of the trained tree */
ntest_test(tree,tset,parameters);

/* save it */
ntree_store(tree,"AFLN");
}
```

The Ntree distribution package contains the following programs:

construct.c constructs a Neural Tree,

train.c trains a Neural Tree,

c_and_t.c constructs and trains a Neural Tree,

ntree2print.c transforms a Neural Tree into a human readable form,

tree2maple.c transforms a Neural Tree into a format that can be used by the Maple[1] program to define, evaluate, and plot the tree in different formats. For this purpose Maple has to be initialized with the definitions from the ntree maple package which is included in the file *ntree.maple*.

An example of a parameter file is given in appendix B.

3 The library

The Ntree library consists of the source files *ntree.c, ctree.c, ntest.c*, and *tset.c* and the include files *ntree.h* for external and *ctree.h* for inter module use. The routines permit the construction of classification or regression trees from data, the training of these trees as adaptive fuzzy logic networks, input and output etc. The parameters which control the functionality of all these routines are collected in a single parameter file.

[1] Maple is a language and system for symbolic and numeric mathematical calculation. It is copyrighted by the University of Waterloo, the authors of Maple, and Waterloo Maple Software. Information can be obtained from the URL: http://daisy.uwaterloo.ca/

3.1 ctree.c

3.1.1 ntree_t *ctree_construct(tset_t *lset, parameter_t *parameters)

This routine implements the CART algorithm according to [1]. The arguments are

1. *lset:* the training vectors which are loaded by *tset_load()* (3.3.1), and
2. *parameters:* the parameters which are loaded by *ntree_parameter_load()* (3.2.5).

The return value is a pointer to the constructed tree. The functionality is controlled by the parameters, see section 3.5.

3.2 ntree.c

3.2.1 void ntree_train (ntree_t *ntree, tset_t *lset, tset_t *tset, parameter_t *parameters)

This is the main routine for training an adaptive fuzzy logic tree. The arguments are

1. *ntree:* the tree to be trained which is either loaded from a file by *ntree_load()* (3.2.3) or constructed by *ctree_construct()* (3.1.1),
2. *lset:* the training vectors which are loaded by *tset_load()* (3.3.1),
3. *tset:* the vectors which are used to check the performance of the AFLN after each training epoch. These are only used if the parameter file includes the line:

   ```
   test after each epoch = TRUE
   ```

   Otherwise NULL may be given as *tset*, and
4. *parameters:* the parameters which are loaded by *ntree_parameter_load()* (3.2.5).

The functionality is controlled by the parameters, see section 3.5.

3.2.2 void ntree_store (ntree_t *ntree, char *filename)

Save a tree to a file in a format (preorder, one line per node/leaf) that can be read by *ntree_load()* (3.2.3). The arguments are

1. *ntree:* the tree to be saved, and
2. *filename:* the name of the file.

3.2.3 ntree_t *ntree_load (char *filename, parameter_t *parameters)

Load a tree from a file that was written by *ntree_store()* (3.2.2). The arguments are

1. *filename:* the name of the file, and
2. *parameters:* a pointer to a parameter structure which was loaded by *ntree_parameter_load()* (3.2.5).

The return value is a pointer to the tree.

3.2.4 void ntree_print (ntree_t *ntree, char *filename, parameter_t *parameters)

Save a tree to a file in a human readable format (inorder, one line per node/leaf with indentation). The arguments are

1. *ntree:* the tree to be saved,
2. *filename:* the name of the file, and
3. *parameters:* a pointer to a parameter structure which was loaded by *ntree_parameter_load()* (3.2.5).

3.2.5 parameter_t *ntree_parameter_load (char *filename)

Load the parameters from a file. The arguments are

1. *filename:* the name of the file.

The return value is a pointer to the parameters.

3.2.6 eval_t ntree_eval (ntree_t *ntree, vec_t vec)

Calculate the value that a tree returns for a given attribute vector. The data type of a classification or regression tree and an AFLN is the same. A classification or regression tree is simply an AFLN with all softness radii equaling zero. The arguments are

1. *ntree:* the tree and
2. *vec:* the vector.

The return value is the calculated value.

3.2.7 void ntree_free (ntree_t *ntree)

Free the memory that was allocated while building up or loading the tree. The arguments are

1. *ntree:* the tree to be freed.

3.3 tset.c

3.3.1 tset_t *tset_load (char *filename, parameter_t *parameters)

Load evaluated attribute vectors from a file. The arguments are

1. *filename:* the name of the file, and
2. *parameters:* a pointer to a parameter structure which was loaded by *ntree_parameter_load()* (3.2.5). It contains the dimension of the attribute space.

The return value is a pointer to a structure which contains the evaluated attribute vectors.

The format of the file is as follows: Each line contains one evaluated attribute vector consisting of n digital numbers (real or integer) for the attribute values and one digital number for the evaluation. Lines that begin with # are ignored as well as all strings that don't begin with a digit or a $-$.

3.3.2 void tset_store (char *filename, tset_t *tset)

Save a set of evaluated attribute vectors to a file in a format that can be read by *tset_load()* (3.3.1). The arguments are

1. *filename:* the name of the file, and
2. *tset:* a pointer to a structure which contains the vectors.

3.4 ntest.c

3.4.1 void ntest_test (ntree_t *ntree, tset_t *tset, parameter_t *parameters)

Check the classification or regression performance of a tree. The performance is either measured in the percentage of correctly classified vectors or in mean square and mean absolute error. The arguments are

1. *ntree:* the tree,
2. *tset:* a pointer to a structure which contains the vectors, and
3. *parameters:* a pointer to a parameter structure which was loaded by *ntree_parameter_load()* (3.2.5).

3.5 The parameter file

The parameter file contains all domain specific information and controls the functionality of *ctree_construct()* (3.1.1) and *ntree_train()* (3.2.1).

3.5.1 Parameters concerning several routines

First of all, the dimension of the attribute space must be specified in the parameter file by including:

```
dimension = n
```

This is the only parameter which has no default value and must be set.

By default the task is assumed to be a regression problem in a n-dimensional space with real valued attribute values. For each attribute of symbolic type include a line of the form

```
type i = j
```

in the parameter file. This indicates that the ith dimension is of symbolic type and that the set of possible values consists of $\{0, 1, \ldots, j-1\}$. The dimensions count from 0 upwards. If the task is a classification problem, include the line

```
regression = FALSE
```

In this case you have to specify the evaluations corresponding to the two classes:

```
min_eval = a  ( for the first class )
max_eval = b  ( for the second class )
```

a must be smaller than b and $a = 0$ and $b = 1$ are the default values.

3.5.2 Parameters concerning ctree_construct ()

For classification the error measure which is used in *ctree_construct()* (3.1.1) is the percentage of falsely classified vectors rather than the mean square error.

For regression the impurity measure used in *ctree_construct()* is the mean square error of the samples in a node in relation to their median value. For classification the impurity measure is the Gini index [1] which for a node with median value v_k and a and b as defined above corresponds to

$$(v_k - a)(b - v_k)\,.$$

If you specify in the parameter file

```
mid_factor = m
```

m being an integer value the purity gain for a split is multiplied by

$$\left(1 + \frac{k_l \cdot k_r}{k^2}\right)^m$$

where k is the number of training vectors in the node to be split, k_l is the number of vectors which belong to the left side of the split and k_r the number of vectors which belong to the right side. The bigger m is the more splits dividing a set into nearly equal parts are preferred. The default value is $m = 0$.

By default *ctree_construct()* performs ten fold cross validation. You can choose the cross validation factor by including the line

```
cv_factor = a
```

If $a = 0$ no cross validation is performed but simple cost complexity pruning with 70 per cent of the training vectors used for the construction of a maximal tree and 30 per cent used for pruning. If you include the line

```
prune = FALSE
```

pruning doesn't take place at all and *ctree_construct()* just constructs a maximal tree from the training vectors.

In cost complexity pruning with cross validation several nesting chains of trees are constructed. The main chain is constructed from all learning data. The extra chains constructed by part of the learning data are used to estimate the error for the trees in the main chain. Include

```
print_main_chain = TRUE
```

to obtain information about the error and the size of the trees in the main chain and

```
print_all_chains = TRUE
```

to obtain information about all chains.

By default *ctree_construct()* constructs an AFLN with softness radii just maximally big, so that the evaluation of the regression or classification tree returns the same values as the corresponding AFLN for all vectors in the training set. If you include the line

```
radius_factor = a
```

each of these softness radii is multiplied with a. Here a can be any positive real value. If $a = 0$ is set a pure classification or regression tree with all softness radii set to zero is obtained.

By default *ctree_construct()* with cost complexity pruning chooses the smallest tree with minimal error. Since the error is further reduced afterwards by neural adaptation (the AFLN algorithm) the actual version of *ctree_construct()* can be forced to choose smaller trees. If the line

```
error_factor = a
```

is given, the smallest tree with error factor not bigger than `a` times the minimal error is constructed. A small error factor produces larger trees and a large one returns smaller trees. The default value for the error factor is 1.

3.5.3 Parameters concerning ntree_train ()

First of all, you can choose which parameters in the AFLN should be regarded as free and which parameters remain fixed. By default no parameter is regarded as free. You can include one or several of the following lines:

```
train radius
train split_value
train set
train evaluation
```

For each inner node with a split with respect to an ordered attribute value *train radius* declares the softness radius and *train split_value* the split value as free parameter. For each inner node with a split with respect to a symbolic attribute value *train set* declares the parameters which encode the division of an attribute set into two fuzzy subsets as free. Finally, for each leaf *train evaluation* declares its evaluation as free.

By default *ntree_train()* applies the backpropagation algorithm [7] to the AFLN. If you include

```
algorithm = quickprop
```

in the parameter file, quickpropagation [2] is applied instead. Quickpropagation is much faster than backpropagation but sometimes not as stable. The maximal number of epochs for training is set by including:

```
epochs = a
```

The training will stop earlier if either the mean square error decreased to zero or it didn't decrease by a factor not more than b for m epochs, b being a real and m an integer value. By default $m = 12$ and $b = 1.0001$. You can also set these values by including:

```
stop_after = m
stop_factor = b
```

In each epoch of the training the backpropagation algorithm calculates the gradient of the error function with respect to the free parameters. The step width in euclidean parameter space by which these parameters are adapted is given by:

```
epsilon = s
```

This parameter is highly domain specific and you should try several values in order to get optimal results. It defaults to $s = 0.1$. The step width is adapted automatically by the following heuristic. Increase the step width by a factor of 1.1 if the mean square error decreased during the last four epochs. Decrease the step width by a factor of 1.2 if the mean square error increased two times or alternated two times. If you include the line

```
decrease_epsilon = FALSE
```

the step width is not adapted.

In order to avoid oscillation in each epoch a momentum term is added to the adaptation step. The momentum term defaults to $a = 0.5$ times the previous step. You can set a by including:

```
alpha = a
```

When calculating the partial derivatives with respect to the softness radii and the split values the softness radii appear in the denominator. In order to avoid division by zero or nearly zero values you can set the minimal value for the softness radii to appear in the denominator:

```
min_radius = a
```

It defaults to $a = 0.00001$.

Since the derivative of the error function with respect to the softness radii and the split values is zero if the attribute value is outside of the softness radius in a given node, a small value is added to this derivate in order to give these values a chance to change at all. The value which is added is set by

```
add_to_f_derivate = a
```

and defaults to $a = 0.1$.

The quickpropagation algorithm [2] also calculates the gradient of the error function. The difference to backpropagation lies in the amount of change of the individual free parameters. The idea is to assume a parabolic form for each partial derivative of the error function and calculate its minimum by a quadratic estimate. This algorithm is faster than backpropagation but also less stable. To give it stability the factor by which the current step may exceed the previous one is limited by $a = 1.75$. Include

```
max_growth = a
```

to set this factor to the value a.

If you include

```
test after each epoch = TRUE
```

the AFLN is tested after each training epoch. This testing is done by a second set of evaluated attribute vectors which is handed to *ntree_train()* (3.2.1).

4 Three example applications

In this section we present a selection of benchmark problems to which we have applied our software.

4.1 The MONK's problems

This problem was presented in 1991 during the "Second European Summer School on Machine Learning" [8].

The attribute space consists of a finite universe of 432 robots which are defined by six symbolic attribute values:

$$\begin{array}{lcl} x_0\text{: Head shape} & \in & \{\text{round, square, octagonal}\} \\ x_1\text{: Body shape} & \in & \{\text{round, square, octagonal}\} \\ x_2\text{: smiles} & \in & \{\text{yes, no}\} \\ x_3\text{: holds} & \in & \{\text{sword, balloon, flag}\} \\ x_4\text{: jacket colour} & \in & \{\text{red, yellow, green, blue}\} \\ x_5\text{: wears tie} & \in & \{\text{yes, no}\} \end{array}$$

Three classification tasks are given by boolean expressions which divide the universe into two classes:

- Problem 1: (Head shape = Body shape) or (jacket colour is red).
  For this problem a training set of 124 correctly classified robots is given.

- Problem 2: Exactly two of the six attributes have their first value.
  For this problem a training set of 169 correctly classified robots is given.

- Problem 3: (Jacket colour is green and holds sword) or (Jacket colour is not blue and Body shape is not octagonal).
  For this problem a training set of 122 robots is given of which 5% are incorrectly classified.

Problem	Number of nodes	Number of epochs	CART	AFLN
1	31	7	91.67%	100%
2	69	1000	75%	89.35%
3	5	10	97.22%	97.22%

Table 1: The results of applying the AFLN algorithm to the MONK's problems: In the column labeled CART contains the classification performance of the untrained trees and the column labeled AFLN that of the trained trees.

The task in all of the three problems is to induce a classificator from the training data and measure its performance on the whole universe. The results of applying the AFLN algorithm to these problems are presented in table 1.

The AFLN algorithm has an ability to change the inner structure of a decision tree which cost complexity pruning has not. The tree in figure 1 was generated by applying our CART implementation to the training set of the first MONK problem. It classifies 91.67% of the test set correctly. If you have a close look at it, you can see that the split in node $*$ makes no sense regarding the definition of the two classes. All vectors which go to the left child of this node are classified correctly and vectors which go to the right side might be classified falsely and contribute to the misclassification rate.

If node $*$ could be replaced by its left descendant $**$ the resulting correct classifier would be the minimal one for this problem. But this can not be achieved by pruning, because pruning can only change node $*$ to a leaf which would not result in a correct classifier. The AFLN algorithm on the other hand is able to achieve this effect. To make this clear we only declared the set parameters as free and trained the classification tree with the same training set it was constructed with. After the training all vectors which pass through node $*$ are directed to the left side and thus the resulting classifier is correct. And after cutting off the right subtree of node $*$ a minimal size classification tree for problem 1 is achieved.

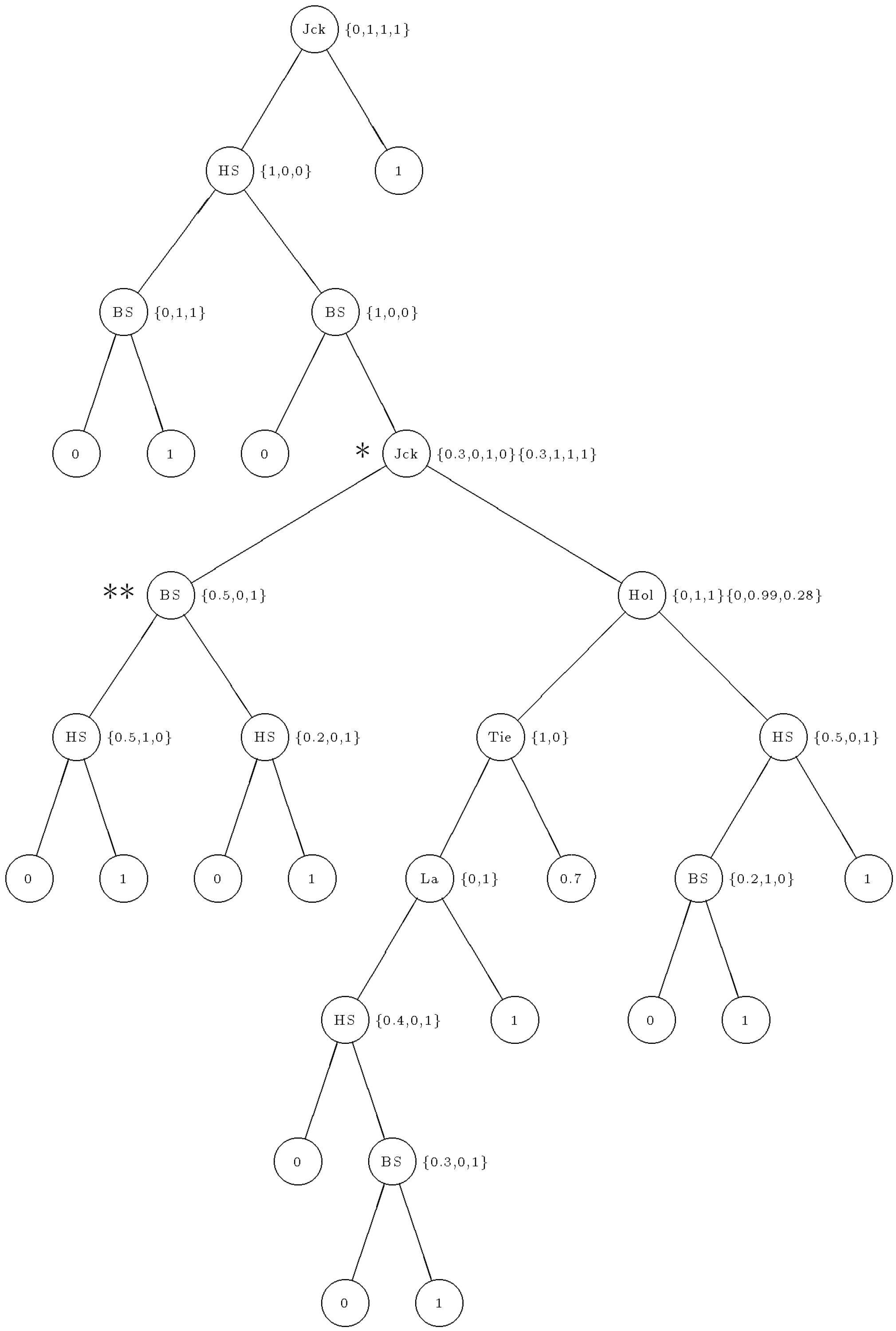

Figure 1: Tree for MONK problem 1: On the right side of the inner nodes we have encoded the division of an attribute into fuzzy subsets. For instance Jck {0,1,1,1} can be interpreted as: If Jacket colour is red go to the right child (0 as first value) and for all other colours go to the left child (1 for all other colours). The leaf evaluations are given in the center of the leaf nodes. The tree was trained with only the subsets as free parameters. The values that have changed after the training are given as a second set of values on the right side of the inner nodes.

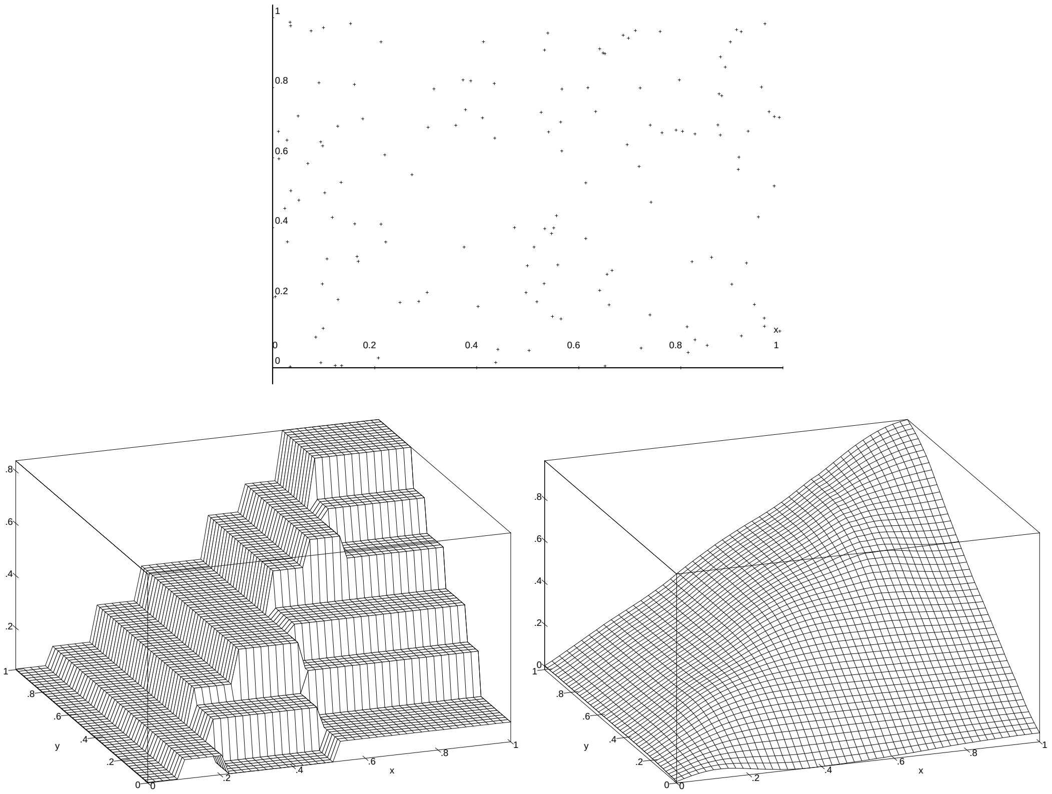

Figure 2: Construction of an AFLN for the MIN function. Upper center: randomly chosen training set. Lower left: function resulting from the CART decision tree. Lower right: function represented by the trained AFLN.

4.2 The minimum function

The task here is to learn the minimum function $min: [0,1] \times [0,1] \to [0,1]$. We have randomly chosen 140 function values for constructing a regression tree and training it and 200 other values for testing the performance. First we have applied 4-fold cross validation in constructing a regression tree with 27 nodes. Then we have trained it for 600 epochs with the quickpropagation algorithm. The results are illustrated in figure 2. The mean square error of the original regression tree of $1.25 \cdot 10^{-3}$ decreased during the training down to $4.63 \cdot 10^{-5}$. This example demonstrates the smoothing properties of the AFLN algorithm.

4.3 The two spirals problem

This problem due to Alexis Wieland constists of two revolving spirals representing elements of two classes. The task is to learn a classificator for the two classes. In this test we have used 385 training points per spiral. In constructing the original classification tree we didn't apply cross validation but used 70% of the training vectors for building up the original tree and the other 30% for pruning. The softness

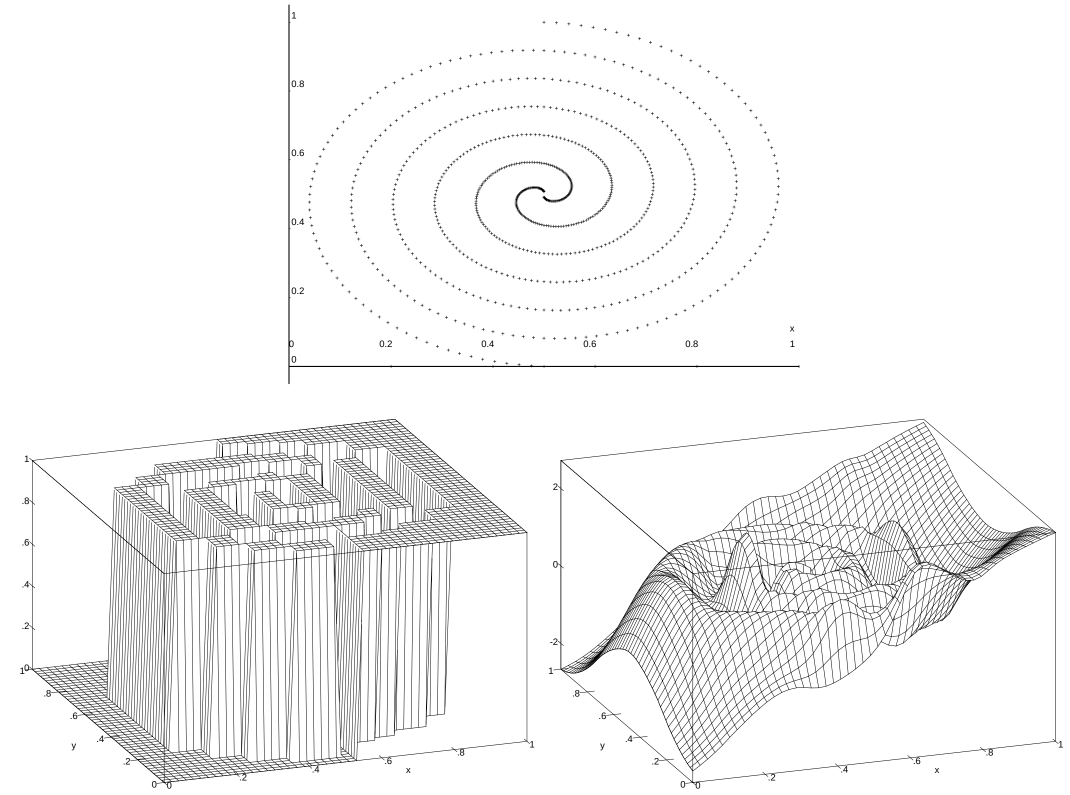

Figure 3: Construction of an AFLN for the two spirals problem. Upper center: The training set. Lower left: function resulting from the CART decision tree. Lower right: function represented by the trained AFLN.

radii were initialized to zero. Then we trained the classification tree for 100 epochs with quickpropagation using all of the training vectors. The results are illustrated in figure 3. The classification performance measured by the 40-20-40 criterion increased from initially 93.35% to 100% within 77 epochs.

This is a hard problem for the CART algorithm because the regions in the attribute plane belonging to one class are not easily approximated with axis parallel rectangles. The AFLN algorithm interpolates the two spirals.

A ntree.h

This is a listing of the include file *ntree.h* which defines the interface to the library:

```
/****************************************************************************
 ****                                                                   ****
 **** Type definitions:   ****
 ****                                                                   ****
 ****************************************************************************/
```

```
typedef double comp_t;    /* type of attribute */
typedef double eval_t;    /* type of vector evaluation */
typedef int bool_t;       /* only TRUE, FALSE, UNEVALUATED or ERROR used */
typedef comp_t *vec_t;    /* type of unevaluated attribute vector */

typedef struct tset {          /* type of training set: evaluated vectors */
     int nu_vec;               /* number of vectors */
     int dimension;            /* number of components of a vector */
     vec_t *vectors;           /* field of vectors */
     eval_t *evaluations;      /* field of evaluations */
     eval_t sum;       /* sum of evaluations */
     eval_t sqr_sum;       /* sum of squared evaluations */
   } tset_t;

typedef struct node_delta{  /* structure to contain extra information */
    /* only of intern use */
    ...
} node_delta_t;

typedef struct leaf_delta{ /* structure to contain extra information */
   /* only of intern use */
    ...
       } leaf_delta_t;


typedef union ntree {   /* node type for AFLN and regression trees */

  union ntree_node {
    union ntree_node *next;  /* for free list management */
    struct {
      bool_t tag;            /* !LEAF for inner node */
      node_delta_t *deltas;  /* additional variables for training */
      int dimension;         /* the dimension by which this node is split */
      int type;          /* the type of this dimension:
   0 = ordered   or
   the size of the set if type is cardinal */
      /* for attributes of ordered type: */
      comp_t split_value;    /* the split value */
      comp_t radius;      /* the softness radius */
      /* for attributes of symbolic type: */
      comp_t *set;      /* array encoding the two subsets to which
this node is split */
      union ntree *child[2]; /* pointer to the two successor nodes */
    } data;
  } node;
```

```
  union ntree_leaf {
    union ntree_leaf *next;  /* for free list management */
    struct {
      bool_t tag;            /* LEAF for a leaf */
      leaf_delta_t *deltas;  /* additional variables for training */
      eval_t evaluation;     /* the evaluation of this leaf */
    } data;
  } leaf;

} ntree_t;

typedef struct parameter {   /* type to contain all parameters */
     int nu_epochs;
     bool_t regression;
     eval_t min_eval;
     eval_t max_eval;
     comp_t min_radius;
     comp_t add_to_f_derivate;
     comp_t alpha;
     comp_t epsilon;
     int algorithm;
     bool_t divide_d_by_nu_vec;
     int nu_vec;
     bool_t train_split_value;
     bool_t train_evaluation;
     bool_t train_radius;
     bool_t train_set;
     bool_t test_after_each_epoch;
     comp_t max_growth;
     comp_t threshold;
     bool_t decrease_epsilon;
     bool_t print_header;
     comp_t eta;
     int stop_after;
     comp_t stop_factor;
     int cv_factor;
     int dimension;
     int *types;
     comp_t mid_factor;
     int test_percentage;
     bool_t prune;
     bool_t print_main_chain;
     bool_t print_all_chains;
     int criterion;
     comp_t radius_factor;
```

```
     comp_t error_factor;
     char **names;
   } parameter_t;

typedef union ntree_node ntree_node_t;
typedef union ntree_leaf ntree_leaf_t;


/* Prototypes of public routines */

void ntree_free(ntree_t *tree);
void ntree_print(ntree_t *tree,char *filename,parameter_t *parameters);
eval_t ntree_eval(ntree_t *tree,vec_t vec);
void ntree_train(ntree_t *tree,tset_t *lset,
tset_t *tset,parameter_t *parameters);
void ntree_store(ntree_t *tree, char *filename);
ntree_t *ntree_load(char *filename,parameter_t *parameters);
tset_t *tset_load(char *filename,parameter_t *parameters);
void tset_store(char *filename, tset_t *tset);
void ntest_test(ntree_t *tree,tset_t *tset,parameter_t *parameters);
parameter_t *ntree_parameter_load(char *filename);
ntree_t *ctree_construct(tset_t *lset, parameter_t *parameters);
```

B A sample parameter file

In order to give the reader an example of how a parameter file looks like, we list a parameter file for the first MONK problem (see 4.1):

```
#*******************************************************
# The parameter file:
# lines that begin with # are ignored
#*******************************************************

#*******************************************************
# decide whether the values are printed while loading
# default: print_header = TRUE
#*******************************************************

print_header = FALSE

#*********************************************************************
# Set the dimension of the feature space.
# This value must be set!!!!!
# It is the only one without default value.
#*********************************************************************
```

```
dimension = 6

#*********************************************************************
# By default attributes are of ordered (real) type.
# If attributes are of symbolic type, specify the size of the set.
# The symbolic values are denoted by 0,1,2,... .
# If for instance the first attribute takes on values out of
# {0,1,2,3} you have to specify:   type 0 = 4
#*********************************************************************

type 0 = 4
type 1 = 4
type 2 = 3
type 3 = 4
type 4 = 5
type 5 = 3

#*********************************************************************
# specify names for the attributes.
# They are used in ntree_print()
#*********************************************************************

name 0 = head-shape
name 1 = body-shape
name 2 = laughing
name 3 = holds
name 4 = jacket
name 5 = tie

#*********************************************************************
#  decide which algorithm to use:  Backpropagation or Quickpropagation
#  default: algorithm = backprop
#*********************************************************************

# algorithm = backprop
algorithm = quickprop

#***************************************************
#  decide how many epochs the tree is to be trained:
#  default: epochs = 10
#***************************************************

epochs = 10

#***************************************************
```

```
# the training is stopped before _epochs_ if
# _stop_after_ many epochs the mean square error
# didn't decrease by more than _stop_factor_
# default: stop_after = 0  ( don't stop before _epochs_)
####################################################

stop_after = 12
stop_factor = 1.0001

#######################################################
#  decide which parameters in the tree shall be trained:
#  by default there are no free parameters, so you have
#  to uncomment at least one of the following
#######################################################

train radius
train split_value
train evaluation
train set

##########################################
#  some parameters for the training process:
##########################################

epsilon = 0.1  (the step width)

decrease_epsilon = TRUE
#decrease_epsilon = FALSE   (if set to TRUE, epsilon is lowered during
#      the training for stability)

alpha = 0.5     (momentum term)

min_radius = 0.00001     (minimal softness radius:
# used in divisions )

add_to_f_derivate = 0.1     (amount added to the derivate
#      of the transfer function)

max_growth = 1.75     (for quickpropagation:
#      the maximal factor by which
#      the current step may exceed
#      the previous step)

threshold = 0.00000002     ( for quickpropagation )

############################################
```

```
#  Some settings concerning performance tests:
#********************************************

#test after each epoch = TRUE
test after each epoch = FALSE

#**********************************************
# By default a regression tree is constructed
# If a classification tree is to be constructed
# specify:  regression = FALSE
# and the values by which the vectors
# are classified, for instance
#
# min_eval = 0  ( for the first class )
# max_eval = 1  ( for the second class )
#
# In testing the trees the 40-20-40 criterion is applied:
# tree values in [ 0 , 0.4 ] are regarded as class 1,
# values in [ 0.6 , 1 ] are regarded as class 2 and
# values in ] 0,4 , 0,6 [ as undecided.
#*********************************************

#regression = TRUE
regression = FALSE

min_eval = 0
max_eval = 1

#*******************************************************
# settings concerning tree construction:
#*******************************************************

#*******************************************************
# The bigger mid_factor, the more splits which divide
# the given set into subsets of nearly equal size are
# favoured.
# default: mid_factor = 0
#   ( choose the best split regardless of the size of
#     the subsets )
#*******************************************************

mid_factor = 4

#*******************************************************
# Choose the cross validation factor:
#
```

```
# cv_factor = 0: no cross validation
# cv_factor = n: n-fold cross validation
#
# default:  cv_factor = 10  ( 10-fold cross validation )
########################################################

cv_factor = 5

########################################################
# print_main_chain = TRUE:
# print information about the main chain of trees
# constructed in the cost-complexity pruning.
# print_all_chains = TRUE:
# print information about all the chains constructed
# in cost-complexity pruning with cross validation
#
# default:  print_main_chain = FALSE
#     print_all_chains = FALSE
########################################################

print_main_chain = FALSE
print_all_chains = FALSE

########################################################
# By default cost-complexity pruning chooses the
# smallest tree with minimal error.
# If error_factor is set, the smallest tree is chosen
# with error not bigger  error_factor times the minimal
# error:
# default:  error_factor = 1
########################################################

#error_factor = 1
```

References

[1] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, California, 1984.

[2] S. E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Computer Science Department, Carnegie Mellon University, Pittsburgh, June 1988.

[3] A. P. Heinz. Efficient implementation of a neural net α-β-evaluator. Technical Report A-1993-2, ISBN 951-45-6592-4, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1993.

[4] A. P. Heinz. Efficient neural net α-β-evaluators. In M. Marinaro and P. Morasso, editors, *Proceedings of the International Conference on Artificial Neural Networks*, pages 635–638, Sorrento, Italy, May 1994. Springer-Verlag.

[5] A. P. Heinz. Fast bounded smooth regression with lazy neural trees. In *Proceedings of the IEEE International Conference on Neural Networks*, volume III, pages 1755–1759, Orlando, Florida, June 1994.

[6] A. P. Heinz. Learning and generalization in adaptive fuzzy logic networks. In H.-J. Zimmermann, editor, *Proceedings of the Second European Congress on Intelligent Techniques and Soft Computing, EUFIT'94*, volume III, pages 1347–1351, Aachen, Germany, Sept. 1994. ELITE, Verlag der Augustinus Buchhandlung.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Vol. 1*, chapter 8. MIT Press, MA, 1986.

[8] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Džeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The MONK's problems — a performance comparision of different learning algorithms. Technical Report CMU-CS-91-197, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Dec. 1991.