Symbolic function network

George S. Eskander a, Amir F. Atiya b,*

a Department of Computer Engineering, Baha Private College of Science, Al Baha, Saudi Arabia
b Department of Computer Engineering, Cairo University, Giza, Egypt

ARTICLE INFO

Article history:
Received 7 April 2007
Revised and accepted 18 February 2009

Keywords:
Symbolic networks
Multilayer networks
Tree networks
Neural tree
Symbolic functions
Computer algebra
Tree propagation
Forward greedy search
Backward greedy search

ABSTRACT

In this paper a model called symbolic function network (SFN) is introduced; that is based on using elementary functions (for example powers, the exponential function, and the logarithm) as building blocks. The proposed method uses these building blocks to synthesize a function that best fits the training data in a regression framework. The resulting network is of the form of a tree, where adding nodes horizontally means having a summation of elementary functions and adding nodes vertically means concatenating elementary functions. Several new algorithms were proposed to construct the tree based on the concepts of forward greedy search and backward greedy search, together with applying the steepest descent concept. The method is tested on a number of examples and it is shown to exhibit good performance.

© 2009 Elsevier Ltd. All rights reserved.

1. Introduction

Traditional neural networks consist of layers of neurons that perform a weighted sum operation where some neuron activation function is applied (such as the log-sigmoid function). While this formulation has proved very flexible and effective for many applications, one cannot deny the fact that in general different functional forms suit different applications and that no specific architecture is best for all. In this paper a novel formulation is proposed with the goal being to impart additional flexibility. The proposed model, called symbolic function network (SFN in short), is based on using elementary functions such as powers, the exponential function, the logarithm as building blocks. These building blocks are used to “build” the final function that models the given application. This is done by combining these building blocks together (and adapting their parameters) in a way so that we achieve a fitting performance as best as possible for the training data.

A relevant research topic is the area of computer algebra (or symbolic computation) which deals with formulas composed of elementary functions/operations and analyzes how to simplify, manipulate and implement them. The goal in this paper is different. It is to synthesize a function made from elementary functions/operations that models a given set of training data points in a regression framework. The term symbolic was chosen because we are working with these functional constituents.

In the proposed approach the function was modeled in the form of a tree, similar to how computer algebra researchers model algebraic formulas. The tree formulation is suitable for the proposed approach because it allows breadth (for example by a sum of many elementary functions) as well as depth in modeling (by concatenations of many functions). In the proposed framework an algorithm was developed to construct the tree and an optimization algorithm was proposed based on the steepest descent concept that determines the parameters of the functional forms in the tree.

The advantage of the proposed approach is that it lets the data dictate what functional forms are pertinent to it, and does not force upon it some unsuitable form. For example an exponentially growing function would more suitably need to be fitted with an exponential function, rather than imposing on it a series of log-sigmoid functions. As a result, the proposed algorithm is expected to potentially produce more concise functional fits. This is useful in applications needing real-time processing speed (in recall), for example prediction of Internet traffic (Atiya, Aly, & Parlos, 2005; Bhattacharya, Parlos, & Atiya, 2003), or packet loss rate (Atiya, Yoo, Chong, & Kim, 2007). Of course traditional neural networks with log-sigmoid activation function have the advantage that they can approximate any continuous function, but this perhaps comes at the price of ending up as a large network.

* Corresponding author.
E-mail addresses: georgeeskander@yahoo.com (G.S. Eskander), amir@alumni.caltech.edu (A.F. Atiya).
There have been some models in the literature that model neural networks in the form of a tree. For example, the Competitive Neural Tree model (CNeT) (Behnke & Karayiannis, 1998) is a multilevel competitive neural tree network for pattern classification. During learning, all nodes of each level compete for each input. The connection weights of the winner are then updated using gradient descent learning. Neural Tree Network model (NTN) (Sankar & Mammon, 1991a, 1991b, 1993) is a multilayer neural tree network for pattern classification. The network consists of single layer neural networks connected in a tree structure. The tree grows by a heuristic learning procedure based on the \( L_1 \) norm and the classification error. Zhang et al. (1997) and Zhang (2002) present the concept of sparse neural trees. The model is designed to evolve sigma–pi networks with either a sigmoid type transfer function or a radial basis function (RBF) used in computing the neuron output. The model allows both architecture and parameters learning by genetic search. Sanger (1991) presents a tree-structured network and a learning algorithm that grows the tree so as to approximate a high dimensional function using a minimum number of input features. This work is based on the concept of approximating nonlinear functions as a linear combination of a nonlinear basis. The tree structure is used here to fine-tune the weights of the main linear network by adding to them sub-networks that consist of other linear combination of nonlinear basis. Other approaches include the recursive neural network model (Gori, Kuchler, & Sperduti, 1999) that has some aspect of tree modelling, and the genetic programming approach of Koza (1991), developed for the purpose of discovering the functional form of an economic model and that also uses some kind of tree representation. All these models differ considerably from the proposed approach.

However, the models most related to this work are the following two models that were recently developed and simultaneous to the work presented here. Chen, Yang, Zhang, and Dong (2005) presented the Evolving Additive Tree model (EAT). This model is also a tree structured hybrid model of mathematical operations, linear and nonlinear terms. The structure and weights of the additive tree are evolved by a tree structure based evolutionary algorithm, and a random search algorithm, respectively. The other model is the Flexible Neural Tree model (FNT), developed by Chen, Yang, and Abraham (2005, submitted for publication), Chen, Yang, Zhang, and Dong (2004), Chen, Yang, and Dong (2004), Chen, Yang, Dong, and Yang (2004). It is a kind of irregular multi-layer network that has a tree structure and is being evolved based on a pre-defined instruction/operator set. This allows input selection, and different activation functions. As being an evolutionary nonlinear system, evolving the tree is done through structure and weights optimization. For the structure optimization problem, various methods have been applied, for example genetic programming (Chen et al., 2005, submitted for publication, 2004), ant programming (Chen et al., 2004), probabilistic incremental program evolution (PIPE) (Chen et al., 2004, 2004). For the parameter optimization problem the following methods have been applied: particle swarm optimization (Chen et al., 2005, submitted for publication, 2004), random search (Chen et al., 2004), and a variant of simulated annealing called degraded ceiling (Chen et al., 2004). The work proposed in this paper has distinct differences from the EAT and the FNT models. While the architecture for our method has similarities with EAT and FNT, the proposed construction algorithms and the parameter determination algorithm are different. The construction algorithms proposed here are based on the concepts of forward greedy and backward greedy search approaches. These are concepts that have been of wide use in the feature selection area (Chen, Donoho, & Saunders, 1998; Cotter, Kreutz-Delgado, & Rao, 2002; John, Kohavi, & Pfleger, 1994; Narendra & Fuknaga, 1997; Suykens, Lukas, & Vandewalle, 2000).

Concerning parameter determination, a steepest-descent based algorithm is proposed, and a “tree-propagation” approach is derived to determine the gradient.

The paper is organized as follows. Section 2 describes the proposed method. In Section 3 the simulation results are presented. Some comparison results with the related methods are presented in Section 4. Section 5 gives some comments on the results, followed by the conclusion section.

2. The proposed method

2.1. Overview

Representing a function in a symbolic form in terms of a number of elementary functions (for example powers, exponential functions, sinusoids, etc) and elementary operations (for example +, − , *, /) in standard computer algebra work is typically accomplished in the form of a tree representation (Cohen, 2002). Taking cue from these approaches, our proposed model is in the form of a tree that is built in a constructive way in a top down fashion. Elementary functions are added to the tree one by one in some way so that we achieve a fitting performance as best as possible for the given training data. Fig. 1 shows an example of a constructed tree. Every node represents some elementary function applied to the sum of variables/functions associated with its child nodes. Each terminal node represents some input variable. By having several layers of the tree, several levels of function concatenations can be achieved.

Three basic elementary functions are considered: powers, the exponential function and the logarithm. The reason for choosing these is the prevalence of these three functions in models of many real-world problems, so their use is expected to cover a lot of ground. Since \( \log(u) \) is not defined for non-positive \( u \) and since \( u^v \) might not be defined for negative \( u \) or for \( u = 0 \) for some values of \( v \), modifications of these functions have been used to avoid getting trapped in non-defined regions. The final three elementary functions are the following:

\[
E_1(u) = u/(u^2 + 1)^\alpha \\
E_2(u) = q e^{au} \\
E_3(u) = p \log(u^2 + 1)
\]

where \( u \) is the input argument and \( v, \nu, q, \alpha, \) and \( p \) are parameters or weights.

Consider again Fig. 1, and let us now interpret the construction of the tree. Let \( u_1, u_2, \ldots \) be the input variables. Each leaf node denotes an input variable and the link above it gives the elementary function to be applied to it (such as \( E_1, E_2, \) or \( E_3 \) of Eq. 1). The function modelled is given by (ignore for now the dashed links in the figure):

\[
Y = E_1(u_2) + E_1(u_1) + E_3(u_2) + E_1(u_2) + E_2(u_1 + E_3(u_2)).
\]
Training is performed in an incremental fashion. For example, assume that during the training process adding the dashed links in the figure is considered. After adding the links the network function will be

\[
Y = E1(u_2) + E1(u_1) + E3(u_2) + E1(u_2 + E2(u_1)) + E3(u_2 + E2(u_1 + E3(u_2))).
\]

Let the original error before adding the link be \( \text{Eold} \). After adding the link, the steepest descent algorithm is applied to adjust all the network weights (all parameters \( w, v, q, \alpha, \) and \( p \) for all nodes), not just the ones that correspond to the added link. Let the new error be \( \text{Enew} \). If \( \text{Enew} < \text{Eold} \) a where \( a \) is a positive threshold then this added link will be kept. Otherwise, it will be discarded and the old configuration will be restored.

Note that for each link there is a baseline input variable on which it operates. For example at the parent node where the new link is added it used to be \( E1(u_2) \), i.e. the baseline variable is \( u_2 \). After adding the link the baseline variable is kept in addition to the added link, so that \( E1(u_2) \) is replaced by \( E1(u_2 + E2(u_1)) \). The reason for that is that by adding a link it is better to not "disrupt" much of the overall function of the network. By keeping \( u_2 \) it is like the old function plus some added term. At least theoretically if the multiplier weight \( w, p, \) or \( q \) of the new link is zero then we get the same error performance as the old network, thus having possibly a smooth transition when adding the link.

### 2.2. Computation of the gradient

The proposed training algorithm is based on the steepest descent concept. Therefore it is needed to compute the gradient of the error w.r.t. the network weights.

In the following the proposed algorithm is presented for computation of the gradient. For illustration consider Fig. 2.

Consider a particular path along the tree: \( Y = Z1 - Z2 - Z3 \) where the functions encountered along the path are: \( E_{i1} - E_{i2} - E_{i3} \)

Then, \( z_{j-1} = E_{ij}(z_j) + E_{oth}(z_{oth}) \) \hspace{1cm} (2)

where \( E_{oth}(z_{oth}) \) is the sum of other transformations that are affecting \( z_{oth} \) which are the siblings of \( z_j \).

Let the error function be

\[
J = \sum_{m=1}^{M} (y(m) - d(m))^2 = \sum_{m=1}^{M} e(m)^2
\]

where \( y(m) \) and \( d(m) \) are the actual network output and the target output for training example \( m \) respectively, and \( M \) is the size of the training set.

The instantaneous gradient w.r.t some weight \( \omega \) is

\[
\frac{\delta J}{\delta \omega} = 2e(m)\frac{\delta y(m)}{\delta \omega}.
\] \hspace{1cm} (4)

However the total gradient for the whole training set is

\[
\frac{\delta J}{\delta \omega} = 2 \sum_{m=1}^{M} e(m)\frac{\delta y(m)}{\delta \omega}.
\] \hspace{1cm} (5)

To obtain \( \frac{\delta y}{\delta \omega} \) (the index \( m \) is skipped for ease notations), a key quantity has to be evaluated, namely \( \frac{\delta y}{\delta z} \). It is obtained using the chain rule by starting up the tree with \( (j = 1) \), and tracing the tree going downward as follows:

\[
\frac{\delta y}{\delta z_j} = \frac{\delta y}{\delta z_{j-1}} - \frac{\delta y}{\delta z_j}.
\] \hspace{1cm} (6)

This partial derivative of the network output w.r.t any hidden node output can then be evaluated in a recursive manner. Using Eqs. (2) and (6) we get:

\[
\frac{\delta y}{\delta z_j} = \frac{\delta y}{\delta z_{j-1}} E'(z_j).
\] \hspace{1cm} (7)

where \( E'(z_j) \) is obtained by differentiating the basic functional forms Eq. (1) w.r.t its argument \( u \) and evaluated at \( u = z_j \).

Once \( \frac{\delta y}{\delta z_j} \)'s are evaluated for all the tree, the gradient can be obtained.

Let \( \omega \) be the weight associated with the node, i.e.

\[
\frac{\delta J}{\delta \omega} = \frac{\delta J}{\delta z_j} + \frac{\delta E_{\omega}}{\delta \omega}.
\] \hspace{1cm} (8)

\[
\frac{\delta y}{\delta \omega} = \frac{\delta y}{\delta z_{j-1}} \frac{\delta E_{\omega}}{\delta z_{j-1}}.
\] \hspace{1cm} (9)

\[
\frac{\delta E_{\omega}}{\delta \omega} \text{ can be evaluated by differentiating the basic functional forms Eq. (1) w.r.t the weights } w, v, q, \alpha, p \text{ whatever } \omega \text{ represents.}
\]

### 2.3. Structure optimization

Structure optimization deals with the issue of constructing the tree; that is determining the strategy and sequence of node additions. Several algorithms are proposed here for structure optimization. These algorithms are described as follows:

#### 2.3.1. Forward algorithm

In the forward algorithm, the network components are added to the network in an incremental way. The network component can be a single link or a complete layer.

Adding a complete layer means adding simultaneously one new node child for each possible function and for each input variable to all leaves (up to that point) of the tree. The algorithm builds the network incrementally from top to bottom. It starts with an empty network then adds the network elements one by one. After adding each element and training the network, its performance is measured and based on whether it is decided to keep this added item or to restore the previous network configurations. The following are the steps of the proposed algorithm:

Algorithm steps:

1- Start from a blank network. Initialize all weights to zero.

2- Test the effect of adding a link (a layer) on the network performance, as follows:

2-a- Store the current network configuration.

2-b- Add a link (a layer) to the network by selecting the link (layer) terminating node types. Then initialize its associated weights to small random values (the weights for the nodes already existing in the network are initialized by their original values, i.e. the values they currently have.)

2-c- Run the optimization algorithm on the updated network scheme (by updating all network weights) and find the resulting network performance.
2-d- Decide to keep the new added link (layer) or to restore the network to the old scheme based on comparing the performance improvement to a threshold set by the designer.

3- Repeat step 2 until one tries all possible elementary functions/input variables or achieves the performance goal.

**Forward model with reduced random set capability:**

When the number of features \( (N) \) is too large, then each layer in the network would grow by too much as we go down the tree. Call the total number of candidate links for any layer \( (L) \) the complete band \( (CB) \) where: \( CB(L) = (3 \times N)^L \).

To limit the resulting computational burden the following variant is proposed. Instead of going through all the features one by one, at each step a distinct randomly selected link is being obtained from the available complete band. A parameter called a reduction factor 'RF' is introduced that controls the number of links to be selected at each layer. The reduced Band set \( (RB) \) is given by:

\[
RB(L) = \text{ceiling}(CB(L) \times (RF^L)) = \text{ceiling}(3 \times N \times RF)^L.
\]

2.3.3. **Backward algorithm**

For some applications that involve high nonlinearities it is often not practical to train the network in a forward greedy approach by adding a single link each time. The other option is to apply the forward algorithm by adding a complete layer each time. This approach has an advantage that the network is being constructed and approximates the target in a short time, but it has a disadvantage that the constructed network is not a sparse one and probably contains many superfluous links. For this reason the backward model has been designed. It jumps to a good approximating but not sparse network in a short time – using the forward model by adding a layer each time – but then it applies a pruning algorithm in order to remove the redundant weights from the network. Here are the details of the algorithm:

**Algorithm steps:**

1- Start from a blank network. Run the forward algorithm by adding a complete layer at a time, until reaching the maximum depth or achieving the performance goal.

2- Run the pruning algorithm, as follows:

2-a- Start from the full weights network designed in step 1, and apply pruning in a bottom-up direction starting from the last layer.

2-b- Test the effect of removing a terminal link on the network performance, as follows:

2-b-1- Store the current network configuration.

2-b-2- Remove this terminal link from the network.

2-b-3- Run the optimization algorithm on the updated network scheme and find the associated network performance.

2-b-4- Decide to keep this link removed or to restore the network to the old configuration based on comparing the performance degradation to a threshold set by the designer.

2-c- Repeat step 2-b until testing all terminal nodes.

Note: When applying a pruning step and one node is removed, its parent node could become a terminal node. So next time pruning is applied it could be removed as well. This is the way pruning can possibly ultimately access all nodes, not just the terminal nodes.

2.3.3. **Forward backward algorithm**

By applying the forward algorithm on some regression applications it is noticed that often while the training error is getting better by adding more links, the generalization performance is getting worse. This is due to the well-known “overfitting” effect. So, a model variant has been designed that is based on running a pruning algorithm in parallel with the constructive algorithm. We found that this scenario enhances the generalization performance and solves the problem of “training error — generalization error” trade off. Also, by applying pruning in parallel there is a chance to re-evaluate the already admitted links.

**Algorithm steps:**

1- Start from a blank network. Run the forward algorithm by adding a single link at a time.

2- If the number of consecutively added links reaches \( K \) (\( K \) is a parameter set by the designer), then run the following pruning algorithm:

2-a- Store the current network configurations.

2-b- Remove a link selected from the set of terminal links.

2-c- Run the optimization algorithm on the updated network scheme and find the resulting network performance.

2-d- Restore the network configuration stored in 2-a.

2-e- Repeat steps 2-b-d for the entire candidate links.

2-g- Compare the network performances computed in 2-c, and identify the worst link and remove this link.

3- Repeat step 1 (starting from the current network structure), and step 2 until trying all possible links or achieving the performance goal.

Notes:

1) The backward algorithm resembles the forward–backward algorithm in that both algorithms apply a pruning phase after an incremental phase, but there are some differences. In the backward algorithm the incremental phase uses a complete layer as the network component to be added at each step, while in the forward–backward algorithm, the network component is a single node. Moreover, for the backward algorithm all permissible network layers are being tested before starting the pruning phase. Also, the incremental and pruning phases are not repeated but performed only once. In contrast, in the forward–backward algorithm, the incremental phase continues until reaching a specific number of selected nodes \( (K) \). This number is initially set by the designer depending on the problem complexity and on the network size – network fitness trade-off. The incremental and pruning phases are repeated until achieving the performance goal or reaching the maximum permissible network depth set initially by the designer. Because of all these differences, we introduced each of these two techniques as separate algorithms instead of introducing the backward algorithm as a special case of the forward–backward algorithm.

2) Concerning the choice of the number of layers, it is a trade-off between complexity and capability. At some point adding more layers will lead to a considerably bigger model (contrary to the philosophy of having a concise representation) and might not buy us much extra performance. At this point we know that we have to stop. We therefore have to check the validation performance after each layer addition and weigh the benefits/drawbacks of proceeding further or terminating. Of course there is the maximum permissible network depth that is set initially by the designer, beyond which it is computationally not practical to proceed.

3) Please note that in all these algorithms, after adding or removing a link or a layer we retrain the network (i.e. adjust the old as well as the new weights). We do not freeze the old weights, as this way we forgo the extra flexibility of the network provided by the old weights. These old weights have to be refined to adapt to the changes in the network structure due to adding (or removing) the extra links.
shows the test performance of the SFN and its
illustrates the FRS-SFN (2) (RF
FRS-SFN (3) (RF
FB-SFN (2) (K
FB-SFN (3) (K
FLK-SFN (2)
FLK-SFN (3)
B-SFN (2)
B-SFN (3)
FB-SFN (2) (K = 5)
FB-SFN (3) (K = 5)
FRS-SFN (2) (RF = 0.5)
FRS-SFN (3) (RF = 0.5)

Table 1
Training and testing error (in MSE) for Example 1 for the proposed model variants FLY-SFN, FLK-SFN, B-SFN, FB-SFN, and FRS-SFN each applied on different maximum depth networks, and for some competing methods B-BP, ES-BP, and BR-BP. Also shown are the average number of weights for all resulted networks.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training Error</th>
<th>Testing Error</th>
<th>Average # weights</th>
<th>Training time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best MSE</td>
<td>Worst MSE</td>
<td>Average MSE</td>
<td>Best MSE</td>
</tr>
<tr>
<td>B-BP (6)</td>
<td>0.0025</td>
<td>0.0027</td>
<td>0.0026</td>
<td>0.0011</td>
</tr>
<tr>
<td>ES-BP (6)</td>
<td>0.0026</td>
<td>0.0055</td>
<td>0.0033</td>
<td>0.0033</td>
</tr>
<tr>
<td>BR-BP (12)</td>
<td>2.36E−14</td>
<td>2.94E−11</td>
<td>6.14E−12</td>
<td>1.28E−09</td>
</tr>
<tr>
<td>R-BP (12)</td>
<td>3.00E−05</td>
<td>1.00E−04</td>
<td>1.00E−04</td>
<td>2.70E−04</td>
</tr>
<tr>
<td>FLY-SFN (2)</td>
<td>0.0088</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLY-SFN (3)</td>
<td>1.49E−04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLK-SFN (2)</td>
<td>0.0053</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLK-SFN (3)</td>
<td>0.0045</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-SFN (2)</td>
<td>0.0085</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-SFN (3)</td>
<td>1.16E−4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FB-SFN (2) (K = 5)</td>
<td>9.22E−4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FB-SFN (3) (K = 5)</td>
<td>7.38E−5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRS-SFN (2) (RF = 0.5)</td>
<td>4.68E−04</td>
<td>0.0022</td>
<td>0.0016</td>
<td>4.50E−04</td>
</tr>
<tr>
<td>FRS-SFN (3) (RF = 0.5)</td>
<td>4.38E−04</td>
<td>8.30E−04</td>
<td>6.29E−04</td>
<td>3.38E−04</td>
</tr>
</tbody>
</table>

3. Simulation results

3.1. Synthetic examples

All versions of the proposed method: (1) forward layer by layer (FLY-SFN), (2) forward link by link (FLK-SFN), (3) backward (B-SFN), (4) forward backward (FB-SFN), and (5) forward with reduced random set (FRS-SFN) have been tested on a number of problems that cover a wide range of applications such as function approximation, regression, system identification, and time series forecasting. The data points are partitioned into 75% training set and 25% validation set. In addition, there is a test set for the final test of the performance of the models. The training set is used for the weight optimization process and the validation set is used to evaluate the network structure in the link admission process. To obtain a comparative idea about the performance of the proposed model, we have implemented on these same problems a multilayer perceptron neural network (MLP) (a single hidden layer network).

The following methods for training the MLP have been considered:

1. The basic backpropagation (B-BP): It uses the standard gradient descent with the momentum term.
2. The early stopping backpropagation (ES-BP). It is similar to B-BP in training details, except that the validation data is used in the structure selection – is used in deciding when to stop training.
3. Bayesian regularization backpropagation (BR-BP) (Haykin, 1999). In this approach the cost function is the error function plus a regularization term that penalizes network complexity. This penalty term is based on a Bayesian formulation.
4. Resilient backpropagation (R-BP). In this approach the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. This eliminates the problem that occurs when the gradient has a very small magnitude, and therefore causes training to become too slow.

For all the four training algorithms the epoch update mode is used and set the maximum number of epochs to 10,000. Exploratory runs have been performed to find the best learning rate (η) and the momentum constant (κ), and it is found that the best values over several tested problems are generally 0.05, and 0.2 respectively. So the learning rate and momentum constant are kept fixed at these values for all tested problems. It is well-known that for MLP, the most critical parameter is the number of hidden nodes. We used the validation set to determine this parameter. The networks have been trained with numbers of hidden nodes being 3, 6, 9, 12, 15, and 20, and we selected the one that gives best validation set performance.

Whether SFN or MLP, all of the network structures' performances are evaluated using a validation data set. However, only the winning structure is tested using the test data set and reported. To average out the fluctuations due to the random choice of the initial weights or the randomness of the added noise in some of the problems, for every method five runs have been performed, each run using different initial weights. The best, worst, and average training and testing performances are reported. For the SFN networks and problems that do not have added noise all runs typically lead to the same network (and same performance). In such a case only one run's result is reported. The reason is that for the SFN networks, at every stage of adding a link we initialize the old weights at their ending values, and initialize the newly added weights as zeros (this way we achieve a smooth transition when adding a new link or layer). The error measure used to assess the networks performance is Mean Square Error (MSE); where $MSE = \frac{1}{M} \sum_{m=1}^{M} (y(m) - d(m))^2$ is the size of the network output and the desired output for example m respectively, and M is the size of the data set. Besides the error measures, the number of weights in the resulting networks are given in order to compare the networks' complexities. In the simulation results, the abbreviations of the learning methods are used followed by the number of hidden nodes for the trained MLP network, and by the maximum network depth for the SFN networks. For example ES-BP (6) means an early stopping backpropagation-trained MLP with 6 hidden nodes, while FLY-SFN (3) means a three-layer forward layer by layer SFN.


The first example is a simple one-dimensional function approximation problem. The problem is to approximate the function: $f(X) = \sin(2\pi X)$. Two data sets are generated, each set represents two complete cycles of the sine wave; the first is the training set and consists of 41 samples, X ∈ [−1:0.05:1]. The second set is used for the network testing and also consists of 40 samples, X ∈ [−0.975:0.05:0.975]. The test data points are therefore shifted from the training points by a constant shift that equals 0.025. Table 1 shows the test performance of the SFN and its variants as well as the competing methods. One can see that for SFN good performance is achieved when setting the maximum network depth to a suitable value. The performance is getting better with going to deeper layers. One can observe that the FB-SFN and FRS-SFN methods give a good compromise of sparseness (as measured by the number of weights) and performance. Fig. 3 illustrates the fitting performance for the B-SFN (3) (i.e. a three layer backward-algorithm SFN).

The table also reports training time including the validation and the structure selection process for all methods. One can see that generally SFN is slower than backpropagation but with the gain of
better testing performance AND/OR sparser constructed network. Also, while applying the different versions of SFN, there is a trade-off between the training time and the network size. For example, FLY-SFN (3) results in a large network in a relatively short training time. While the network constructed through FB-SFN (3) ($K = 5$), is much smaller and has a better performance, but with the cost of longer training time.

Fig. 4 illustrates a constructed SFN network. The network is evolved using FRS-SFN (2) (i.e., a two layer forward reduced random set-algorithm SFN). It consists of seven elementary functions and twelve parameters. The performance fitness of the constructed function of the testing set measured in MSE is: $4.50 \times 10^{-4}$.

Fig. 5 illustrates a constructed SFN network. The network is evolved using FRS-SFN (2) (i.e., a two layer forward reduced random set-algorithm SFN). It consists of seven elementary functions and twelve parameters. The performance fitness of the constructed function of the testing set measured in MSE is: $4.50 \times 10^{-4}$.

The network output is given by:

$$Y = E_1(u + E_2(u)) + E_2(u + E_1(u)) + E_3(u + E_2(u) + E_3(u))$$

$$Y = 0.3304(1 + [u - 0.0009e^{4.5983u^2}])^4\cdot 0.6275$$

$$+ 10.0713e^{-4.4452u + 0.3396u^2 + 1.3831}\cdot 0.7065$$

$$\times \log(1 + [u - 1.0049e^{-3.3430u} + 1.832\log(u^2 + 1)])^2.$$  

Example 2. Two-dimension regression

A regression problem is considered in this example, namely approximating a noisy two-dimensional function. Consider the function: $f(X) = X_1^2 + X_2^2 + \varepsilon$; where $\varepsilon$ is a normally distributed random noise with mean $\mu = 0$, and standard deviation $\sigma = 0.05$. The training set consists of 100 samples generated in the grid $[-1 : 0.2 : 0.8] \times [-1 : 0.2 : 0.8]$, and the test set consists of 100 samples generated using the grid $[-0.975 : 0.2 : 0.825] \times [-0.975 : 0.2 : 0.825]$ (i.e., the testing points are shifted from the learning points by a constant shift equals 0.025 in each dimension). Table 2 shows the test comparison results. Fig. 5a shows the noisy data used in training the network, and Fig. 5b shows the SFN network’s fitting performance for the B-SFN (2) (i.e., a two layer backward-algorithm SFN). The results show that B-SFN produces the best performance (beating the MLP methods as well as the other SFN methods). However, FB-SFN (2) and FRS-SFN (2) produce very sparse networks, yet the performance is reasonable.

Example 3. System identification problem

Here the dynamical system (from Atiya and Parlos (2000)) is considered.

$$y(k + 1) = 0.4y(k) + 0.4y(k)y(k - 1) + 0.6u(k) + 0.1.$$
The problem is to design a network that identifies the given system using input–output data. Two sequences of 200 points are generated, each by considering the input signal as an independent random noise (uniform in \([0, 0.5]\)). The first sequence is used for training and the second sequence is used for testing the network. The problem is to design a network that identifies the given system using input–output data. Two sequences of 200 points are generated, each by considering the input signal as an independent random noise (uniform in \([0, 0.5]\)). The first sequence is used for training and the second sequence is used for testing the network.

### Table 3

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training Average # weights</th>
<th>Testing Average # weights</th>
<th>Training time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-BP (9)</td>
<td>6.67E-05 1.83E-04 1.23E-04</td>
<td>1.61E-04 9.41E-04 5.61E-04</td>
<td>46 4</td>
</tr>
<tr>
<td>ES-BP (9)</td>
<td>2.96E-05 2.13E-04 1.42E-04</td>
<td>1.76E-04 6.90E-04 3.21E-04</td>
<td>46 3</td>
</tr>
<tr>
<td>BR-BP (3)</td>
<td>1.64E-07 3.92E-05 8.80E-06</td>
<td>6.42E-05 2.51E-04 1.39E-04</td>
<td>16 0.5</td>
</tr>
<tr>
<td>R-BP (3)</td>
<td>2.88E-05 5.49E-05 2.91E-05</td>
<td>5.35E-05 8.71E-04 3.02E-04</td>
<td>16 3</td>
</tr>
<tr>
<td>FLY-SFN (1)</td>
<td>6.58E-04 8.38E-04 7.38E-04</td>
<td>4.92E-04 7.15E-04 6.14E-04</td>
<td>15 0.1</td>
</tr>
<tr>
<td>FLK-SFN (1)</td>
<td>2.95E-04 4.54E-04 3.77E-04</td>
<td>2.14E-04 4.70E-04 3.42E-04</td>
<td>6 1.2</td>
</tr>
<tr>
<td>B-SFN (1)</td>
<td>6.29E-04 8.02E-04 7.05E-04</td>
<td>4.64E-04 6.79E-04 5.83E-04</td>
<td>6 0.5</td>
</tr>
<tr>
<td>FB-SFN (1) (K = 3)</td>
<td>3.28E-04 4.87E-04 4.04E-04</td>
<td>2.67E-04 4.28E-04 3.26E-04</td>
<td>4 1.5</td>
</tr>
<tr>
<td>FRS-SFN (1) (RF = 0.5)</td>
<td>3.28E-04 4.60E-04 3.84E-04</td>
<td>2.37E-04 3.83E-04 3.17E-04</td>
<td>4 0.5</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training Average # weights</th>
<th>Testing Average # weights</th>
<th>Training time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-BP (9)</td>
<td>0.045 0.083 0.068</td>
<td>0.113 0.272 0.158</td>
<td>46 4</td>
</tr>
<tr>
<td>ES-BP (15)</td>
<td>0.080 0.113 0.088</td>
<td>0.116 0.115 0.128</td>
<td>76 2</td>
</tr>
<tr>
<td>BR-BP (3)</td>
<td>0.102 0.105 0.103</td>
<td>0.085 0.147 0.113</td>
<td>16 0.5</td>
</tr>
<tr>
<td>R-BP (3)</td>
<td>0.095 0.108 0.102</td>
<td>0.074 0.130 0.103</td>
<td>16 2</td>
</tr>
<tr>
<td>FLY-SFN (1)</td>
<td>0.115</td>
<td>0.107</td>
<td>15 2</td>
</tr>
<tr>
<td>FLK-SFN (1)</td>
<td>0.129</td>
<td>0.0937</td>
<td>9 4</td>
</tr>
<tr>
<td>B-SFN (1)</td>
<td>0.122</td>
<td>0.0975</td>
<td>9 7</td>
</tr>
<tr>
<td>FB-SFN (1) (K = 5)</td>
<td>0.133</td>
<td>0.0987</td>
<td>7 10</td>
</tr>
<tr>
<td>FRS-SFN (1) (RF = 0.5)</td>
<td>0.127 0.147 0.137</td>
<td>0.0837 0.0940 0.0884</td>
<td>5 3</td>
</tr>
</tbody>
</table>

### Example 4. Time series prediction problem: forecasting river flow

The problem is to forecast the River Nile flow (Atiya, El-Shoura, Shaheen, & El-Sherif, 1999). The readings of the daily river flow volume in millions of cubic meters during the ten years period from 1985 to 1994 are used. This flow is measured at the Dongola station, located in Northern Sudan (South of high Dam). The single step ahead prediction problem is considered, where the step corresponds to ten-days average flow. To accomplish that, a new time series from the original daily one is created, that is the “ten-day average” time series by taking the average of each ten days flow. The time series is scaled before being used in the forecasting process and then the output of the forecasting model is de-scaled to get the predicted flow volume. The flow data of the period from 1985 to 1989 is used for network training, and the flow data from 1990 to 1994 for testing the networks’ performances. Table 4 shows the comparison results. Fig. 7 shows the prediction versus actual for the testing portion of the time series for the FRS-SFN (1) (i.e. a single layer Forward Reduced Random Set-algorithm SFN). The results show that FRS-SFN method result in the sparsest network and also gives the best performance. One can also see that all SFN methods considerably outperform most of the MLP methods, in both aspects of sparsity and performance.
Example 5. MPEG coded video traffic prediction

Delivering real-time video across the Internet has become a challenging application due to the need to compress the video as much as possible and to cope with network transmission delays and losses. Nevertheless it is an essential element in such advanced services as distance learning, digital libraries, video conferencing, and video on demand. These services could not be delivered across the Internet in a guaranteed quality of service.

This is due to the heterogeneous makeup of the Internet that leads to unspecified available bandwidth for each application, leading to variable transmission delays and quality degradation due to packet loss (Atiya et al., 2005; Bhattacharya et al., 2003). However, real-time video applications are sensitive to time delay, and this effect needs to be mitigated as much as possible. One needed tool for such a step is the prediction of the amount of coded video traffic (at the transmitter side). Predicting the traffic can help in the effective dynamic bandwidth allocation and in multimedia quality-of-service (QoS) control strategies implemented at the network edges. In this latter approach, called "end to end quality-of-service control", the transmission rate is adjusted in a control-theoretical way so as to minimize degradation and give the “streaming” effect needed for real-time video.

We considered here such a problem, that is the prediction of a time-series representing frame or visual object plane (VOP) sizes of an MPEG-coded stream. Needless to say, the prediction (recall) has to be performed in a real-time speed to avoid too much computation overhead, so having a concise network is a beneficial, if not necessary, feature. The time series is extremely noisy, and it has very long-range time dependences. The SFN model is applied on this problem in order to test the model’s ability for such time series prediction problems. We considered the time series of eleven MPEG video files (Fitzek & Reisslein, 1998) used. Each series consists of 89,998 VOPs. The number of I-VOPs in each trace is 7,500, the number of P-frames in each trace is 22,500, and the number of B-frames in each trace is 59,998.

For all trials, NMSE% is used as the performance measure to be minimized where:

\[ \text{NMSE}\% = \frac{\sum_{m=1}^{M} (y(m) - d(m))^2}{\sum_{m=1}^{M} d(m)^2} \times 100; \]

\( y(m), d(m) \) are the network output and the desired output at any sample “m” respectively, and M is the length of the testing set.

Because of the limited space, we report here only the results of the I-VOPs size time series. For the P-VOP and the B-VOP time series we get similar findings. The model is trained using the first 15,000 I-VOP sizes of the data trace of Aladdin, while the designed predictor is tested on the entire video traces of all other video streams. The test error results are shown in Table 5, and sample trace, together with its prediction (for a test portion of the series) is shown in Fig. 8.

Example 6. Packet loss ratio prediction

The quality of multimedia communicated through the Internet is highly sensitive to packet loss. It can severely degrade the quality of especially delay-sensitive multimedia applications (as retransmission of lost packets is not an option). The motivation for predicting the packet loss rate (PLR) arises from two considerations. Firstly, most of the traffic controllers adjust the rate based on the key quantities: packet loss rate and the round trip time (RTT). Rather than using previously measured values of PLR and RTT (i.e. recent values), a better approach is to use predictions of these quantities. The other motivation for predicting PLR is that for real time multimedia traffic transmitted using the UDP protocol, one way to recover lost packets is by adding redundancy using the so-called forward error correction mechanism (FEC). In this approach extra packets are inserted, and these can be used to recover lost packets. The FEC packets, however, represent a bandwidth overhead, and it is therefore a good idea to send only as much of these as needed. The estimate of the packet loss rate is needed to estimate the amount of such redundancy and hence lessening the overhead. Due to the above, an accurate prediction would therefore be very valuable.

The SFN model is considered to design end-to-end packet loss rate (PLR) predictors. The same data found in Atiya et al. (2007) is used; this information is used to estimate PLR. The basic time unit is 2 s, so the time series of PLR values represents the measurement over each 2 s time interval. The resulted test PLR time series consists of 13,158 data points. In designing the SFN model, the first 2000 points are used in training the network and the remaining data as an out-of-samples test data. For all trials, NMSE is used as the performance measure to be minimized. Also; the latest three...
data points are used as inputs to the network in order to predict the next data point. The results are listed in Table 6, and part of the predicted PLR series is represented in Fig. 9.

4. Comparison of the related methods

In this section the goal is to compare the performance of the proposed model with that of two of the most related methods: the Flexible Neural Tree (FNT), and the Evolvable Additive Tree (EAT). It was hard to add FNT and EAT to the compared methods of the previous examples because there was a lot of implementation details for these methods that we did not know. So programming them ourselves will not give them due justice. So to perform such a comparison, we used one example from each of the papers that describe these approaches, applied our methods, and compared with the reported results.

4.1. Flexible neural tree (FNT)

The example in Chen et al. (2004) is considered. For this example the problem is to design a network to approximate a nonlinear static function. The system is described by the equation:

\[ y = (1 + x_1^2 + x_2^{1.5})^2, \quad 1 \leq x_1 \leq 5, \quad 1 \leq x_2 \leq 5. \]

50 training and 200 test samples are randomly generated in the interval \([1, 5]\); Table 7 shows the comparison results.

4.2. Evolvable additive tree (EAT)

The time series prediction problem in Chen et al. (2005) is considered. The problem is to train a network to predict the time series given by:

\[ x(k + 1) = 1.4 - x(k)^2 + 0.3 \times x(k - 1), \quad \text{where } x(k) \in [-2, 2]. \]

100 training and other 100 test samples are randomly generated by randomly selecting the initial conditions of \(x(0)\) and \(x(1)\) in the interval \([-2, 2]\). Table 8 shows the comparison of SFN with EAT. Fig. 10 shows the prediction for the FLK-SFN (2), i.e. a two layer forward link by link-algorithm SFN on the testing portion of the time series.

5. Comments on the results

One can see from the preceding experiments that the SFN methods have been successful in implementing a variety of regression problem, ranging from noise-free function approximation, nonlinear regression, system identification, and time series prediction. Overall, the best variants turned out to be B-SFN, FB-SFN, and FRS-SFN. The B-SFN variant generally leads to best test error. On the other hand, FB-SFN and FRS-SFN lead to a little worse test error but also to a much smaller network (least number of parameters among all MLP and SFN methods). In fact Example 3 yielded a network with only 4 parameters for each of these two methods, and this very compact network still achieved very low error. This is due to the pruning operation performed by the FB-SFN method, and the selective node addition for the FRS-SFN method. Compared to the BP methods, the standard backpropagation MLP (B-BP), the early-stopping MLP (ES-BP), the resilient BP (R-BP), and the Bayesian BP (BR-BP) were generally most of the time worse in performance than B-SFN, FB-SFN and FRS-SFN methods (taking into account the sparseness, i.e. the number of parameters, and the test error). BR-BP was occasionally better than the SFN methods but usually at the
expense of a larger network. So overall, the proposed B-SFN, FB-SFN and FRS-SFN has shown an edge concerning test performance and sparseness. The results compared to other symbolic tree approaches, namely FNT and EAT, were mixed. Often B-SFN, FB-SFN and FRS-SFN were better than its counterpart, and often not.

Concerning speed, the proposed methods are slower to train than the BP approaches. This is due to the fact that training is applied each time we add a link. However, in the majority of applications, it pays to spend the extra computation time in return for improved performance. Also, the fact that we get a more concise network using SFN will lead to smaller prediction (recall) time. This is typically desirable in applications needing real time processing speed (in recall).

6. Conclusion

In this paper, a novel symbolic tree based model is introduced. The goal of this model is to synthesize a function made from elementary functions/operations that models a given set of data points in a regression framework. A tree propagation approach is derived to compute the gradients in a backward fashion, and used to design a steepest-descent based optimization algorithm. Algorithms are designed to construct the tree based on the concepts of forward greedy search and backward greedy search. Simulations have shown good performance compared with the traditional neural networks and some related models.

References


