Using Flexible Neural Trees to Seed Backpropagation

Peng Wu[1(✉)] and Jeff Orchard[2]

[1] School of Information Science and Engineering, University of Jinan, Jinan, China
ise_wup@ujn.edu.cn
[2] Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
jorchard@uwaterloo.ca

Abstract. Neural networks are a powerful computational architecture for modeling data, but optimizing the connection weights can be very difficult. Flexible neural trees (FNTs) are good at finding a globally near-optimal network to fit a dataset, using evolutionary algorithms and particle swarm optimization. We show that putting the two methods together can yield very good results. The FNT solution can be embedded into a larger neural network that is then optimized using backpropagation. The combination of the two methods outperforms either method alone.

Keywords: Neural networks · Flexible neural trees · Backpropagation

1 Introduction

Networks have proven to be a powerful and versatile architecture for modeling data. A variety of techniques and architectures have been developed to train neural networks, including contrastive divergence for Restricted Boltzmann Machines [6], autoencoders [1], convolutional neural networks [7], and backpropagation [8]. However, the optimization problems evoked by these neural network training methods remains – to this day – a fundamental challenge.

The main contribution of this paper is to demonstrate how a flexible neural tree (FNT) can aid in this process. The concise, sparse network generated by the FNT can be used as an initial solution for backpropagation (BP), and yield better performance. Starting BP from a random state often gets caught in local optima, or converges very slowly. Starting the BP with the FNT solution is an effective way to find an optimal neural network solution efficiently.

2 Background

2.1 Flexible Neural Trees

Flexible Neural Trees (FNT) were proposed by Chen [3]; they are a subset of Artificial Neural Network (ANN), restricted to using only a tree structure.

D. Liu et al. (Eds.): ICONIP 2017, Part I, LNCS 10634, pp. 109–116, 2017.
https://doi.org/10.1007/978-3-319-70087-8_12

Compared to ANNs, FNTs have three advantages: (1) the important inputs are automatically selected during its construction procedure; (2) the connections between nodes of two adjacent layers are sparse, which helps avoid overfitting and improve generalizability; and (3) the number of its layers is adaptive to a given training dataset, so the user does not have to decide the layer architecture before training. Benefiting from these advantages, FNTs have achieved a number of outstanding performances in the fields of function regression and pattern recognition [2,4,5].

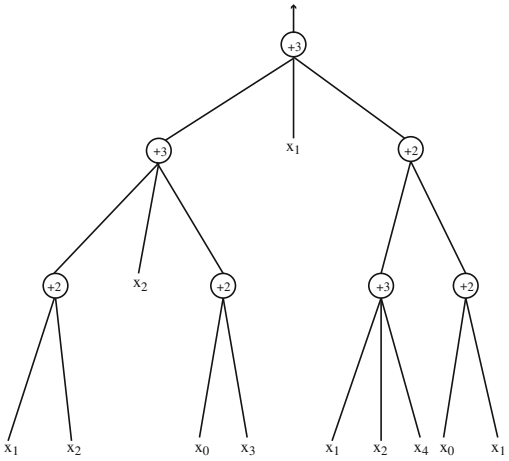

Fig. 1. A flexible neural tree (FNT) model

To construct a FNT model, two sets of nodes are predefined: the non-leaf node set (also called the *non-terminal node set*), and the leaf node set (also called the *terminal node set*). The user can choose different nodes from the two sets to construct a FNT model. Figure 1 shows a FNT model whose nodes are from the non-leaf node set $N = \{+_2, +_3\}$ and the leaf node set $T = \{x_0, x_1, x_2, x_3, x_4\}$. The output of a FNT model is calculated from the leaf nodes to the root node following three rules: (1) the output value of a node will be used as an input value of its connected node; (2) the output of a leaf node is equal to the value of an input variable; and (3) a non-leaf node's output is calculated using

$$y_{\text{non-leaf}} = \sigma\left(\sum_{j=0}^{M} w_j I_j + \theta\right), \tag{1}$$

where I_j denotes the input of the current node, w_j is the corresponding weight, and θ is the node's offset or bias. Three common nonlinear activation functions are

$$\text{Gaussian} : \sigma(x) = \exp\left(-\left(\frac{x-b}{a}\right)^2\right),$$
$$\text{Logistic} : \sigma(x) = \frac{1}{1+\exp(-x)},$$
$$\text{ReLU} : \sigma(x) = \max(0, x).$$

See [4] for a more detailed coverage of the construction of an FNT model.

2.2 Error Backpropagation

Backpropagation is a supervised learning method for training neural networks [8]. Given a neural network, let us denote the action of the network using a single function, $f(x; \Phi)$, where x is the input, and Φ represents the connection weights and biases collectively. Then, given a set of training samples, (x, t), and a cost function $d(y, t)$ that quantifies the mismatch between the network output $y = f(x; \Phi)$ and the target t, neural learning tries to minimize the expected value of that cost,

$$\min_{\Phi} \mathbb{E}\left[d\big(f(x;\Phi), t\big)\right].$$

Example cost functions are the sum-of-squares, and cross entropy.

Backpropagation is based on gradient-descent optimization. Gradient descent requires the gradient of the cost function with respect to each of the network connection weights, Φ_i. Thus, gradient descent incrementally adjusts those parameters in the direction opposite the gradient vector, yielding an update rule

$$\Delta\Phi_i = -\kappa \frac{\partial d\big(f(x;\Phi), t\big)}{\partial \Phi_i},$$

where κ is a scalar learning rate.

3 Methods

The design of our methodology is simple: we attain a FNT model of our data, and use that model as a starting point for BP.

We test our methods on 2 different datasets, which we will call *BC* (for "breast cancer"), and *concrete*. The BC dataset is a classification problem, with 285 training samples and 284 test samples. The concrete dataset is a function regression problem (estimating the output from the input) with 515 training samples and 515 test samples. Both of these datasets are freely available from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml).

In each experiment, we first generate a FNT using genetic programming and particle swarm optimization. Once we have the FNT, we embed it in a neural network. The embedding neural network can either be *minimal* (using only the connections indicated in the FNT), or *full* (with fully connected layers).

We also have two ways of initializing the connection weights and biases: *seeded*, using the connection weights given by the FNT; and *random*, using random connection weights, drawn from a Normal distribution, $\mathcal{N}(\mu = 0, \sigma = 1)$. Thus, we have a total of four different ways of embedding the FNT: *minimal-seeded*, *minimal-random*, *full-seeded*, and *full-random*.

In the *minimal* neural networks, only the connections present in the FNT solution are present, and adjusted by BP. To embed the FNT in a *full-seeded* neural network, we assign a connection weight of zero to any connection that was not in the FNT.

Figure 2 shows the FNT that resulted from the concrete dataset, and Fig. 3 shows the FNT embedded into a neural network with fully-connected layers.

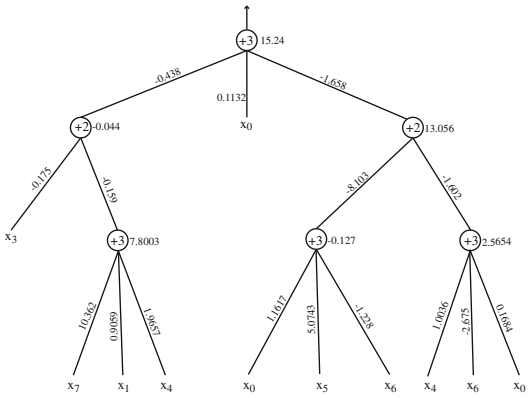

Fig. 2. The FNT model used for concrete dataset.

For the concrete dataset, the nodes of the FNT use the ReLU ("rectified linear unit") activation function, except for the top (output) node, which uses the identity activation function. The same is true for the embedding neural network, with one exception. Notice in Fig. 2 that most of the leaves are at a tree depth of 3, but that one leaf is at a depth of 2, and one leaf is at a depth of 1. In a standard neural network, all the input nodes are at the same depth. To accommodate these different depths, we added "pass-through" nodes to the embedding neural network; these nodes simply relay the value, and thus have an incoming connection weight of 1, a bias of 0, and use the identity activation function.

Figure 4 shows the FNT for the BC dataset. The dataset has 30 inputs, so only a subset of those inputs are actually used by the FNT. The corresponding embedding neural network is shown in Fig. 5. To make the figure more readable, only the nodes used in the FNT are shown. However, the embedding neural network used in our experiments includes all 30 input nodes.

The FNT for the BC dataset also used ReLU activation functions, including the output node. The nodes of the corresponding embedding neural network also

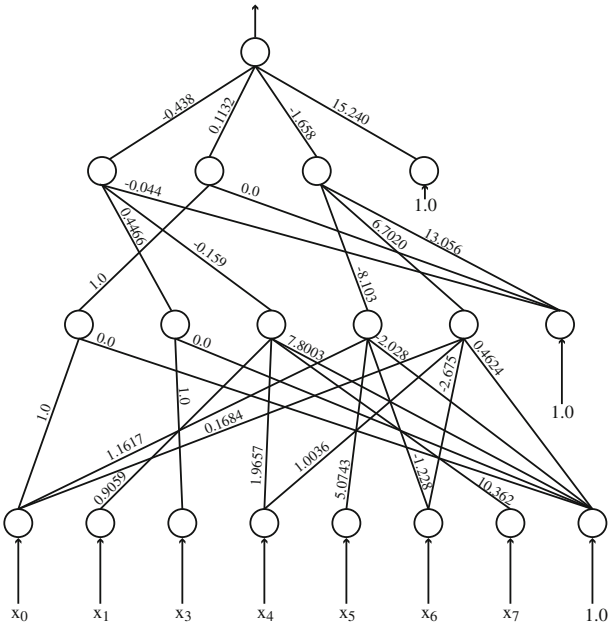

Fig. 3. The FNT model for concrete dataset embedded in a neural network. Any connections not shown in the diagram are assumed to have a weight of 0. At each layer, the bias is depicted as a weighted connection from an additional node that always has a value of 1.

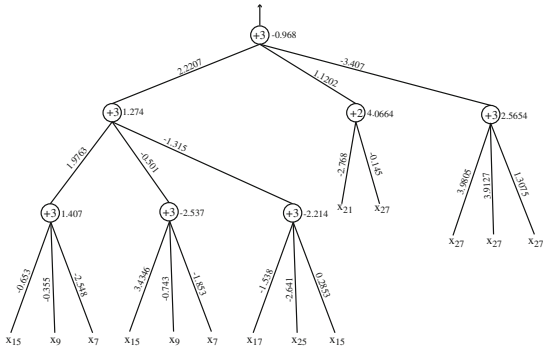

Fig. 4. The FNT model used for BC dataset.

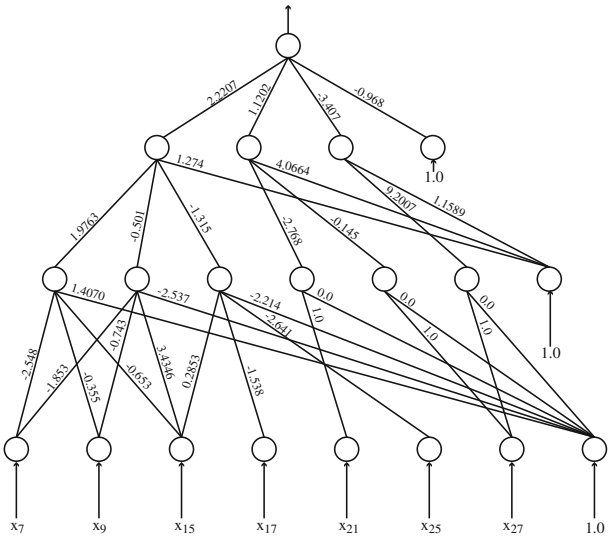

Fig. 5. The FNT model for BC dataset embedded in a neural network. Any connections not shown in the diagram are assumed to have a weight of 0. Note that, for readability, only 7 of the 30 input nodes is shown. The actual embedding neural network used for our experiments has all 30 input nodes.

used ReLU activation functions, with the exception of the pass-through nodes, which used the identity.

Using these embedding neural networks, we applied backpropagation (BP) to adjust the connection weights in an attempt to reduce the output error; RMSE for the concrete dataset, and classification error rate for the BC dataset. For the *minimal-seeded* networks, only the connections corresponding to those present in the FNT were updated using BP; the connections leading into the pass-through nodes were not adjusted. Similarly, for the *minimal-random* networks, only the connections corresponding to those present in the FNT were given random weights; the other connections were kept at 0 (for those not present in the FNT), or 1 (for those leading into pass-through nodes).

For the *full-seeded* networks, all connections were allowed to be adjusted by BP, even those initially set to 0, and those leading into pass-through nodes. Finally, in the *full-random* networks, all connections were initialized randomly and allowed to be updated by BP.

For the concrete dataset, all neural networks were optimized using stochastic gradient descent over 1000 epochs, using a batch size of 10, and learning rate of 10^{-9}. For the BC dataset, networks were also optimized using stochastic gradient descent with a batch size of 10. However, the *minimal* networks were run for 500 epochs using a learning rate of 10^{-2} (to control overfitting), and the *full* networks were run for 2000 epochs using a learning rate of 10^{-5}.

4 Results

The results for the experiments on the concrete dataset are shown in Table 1. The table shows the root mean squared error (RMSE) between the output of the various networks, and the target output from the dataset. As one would expect, the *minimal-seeded* network was able to improve slightly on the FNT model. Moreover, the *full-seeded* network performed the best. These networks were given a very good initial state, seeded by the FNT model, and improved with BP. The additional connections afforded by the *full-seeded* model enabled a slight improvement over the minimal networks.

The random networks (both *minimal-random* and *full-random*) did not perform as well. This indicates that the initial state of the network is a very important factor in the success of BP. Starting near an optimal solution makes a big difference in the outcome, illustrated by the fact that the *minimal-random* networks consistently converged to a non-optimal solution with little variation.

Table 1. RMSE for the concrete dataset

Method	Training	Testing
FNT	9.91 ± 0.687	10.12 ± 1.09
Minimal, Seeded	9.05 ± 0.00056	9.36 ± 0.0018
Minimal, Random	15.2 ± 0.77	15.5 ± 0.68
Full, Seeded	8.57 ± 0.011	8.89 ± 0.027
Full, Random	17.3 ± 2.63	17.7 ± 2.62

The results from the BC dataset are shown in Table 2, which lists the classification error rate for the five network scenarios. This table tells a similar story to the concrete dataset. The FNT model does well, but BP is sometimes able to improve on it slightly; the *minimal-seeded* performs the same as, or slightly better than, the FNT model, and the *full-seeded* network performs the best. Again, we see that the randomly initialized networks were not consistently able to converge to the optimal solution in the allotted time, and exhibited a large amount of variation.

Table 2. Classification error rate for the BC (breast cancer) dataset

Method	Training (%)	Testing (%)
FNT	11.6 ± 2.55	12.4 ± 3.57
Minimal, Seeded	11.7 ± 5.26	8.83 ± 4.88
Minimal, Random	21.4 ± 18.3	19.6 ± 19.3
Full, Seeded	4.74 ± 0.25	6.55 ± 0.56
Full, Random	27.3 ± 19.6	24.6 ± 20.7

5 Conclusions

Each FNT model is a special case from the set of *minimal* networks, and those *minimal* networks are a small subset of the *full* networks. Thus, the globally optimal *full* network should be no worse than the globally optimal *minimal* network, which should be no worse than the FNT model. However, this is not what we observe. The difficulty of the problem is not necessarily articulating a space of solutions. Rather, the difficulty seems to be the process of optimization. Getting BP to converge to the globally optimal solution gets harder and harder as more flexibility is added (in the form of additional connections and nodes).

Using a FNT model as a starting point, we have shown that embedding the FNT model into a neural network allows BP to find an optimal solution quickly and consistently.

Further investigations could study the effect of different activation functions, and different cost functions. It would also be interesting to look at how the size of the embedding neural network affects its performance; does adding more than the minimum number of nodes to intermediate layers improve or worsen performance?

Acknowledgments. This research was supported by the National Key Research and Development Program of China (No. 2016YFC0106000), the Youth Science and Technology Star Program of Jinan City (201406003).

References

1. Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS 2006), pp. 153–160. MIT Press, Cambridge (2006)
2. Chen, Y., Abraham, A., Yang, B.: Feature selection and classification using flexible neural tree. Neurocomputing **70**(1), 305–313 (2006)
3. Chen, Y., Yang, B., Dong, J.: Evolving flexible neural networks using ant programming and PSO algorithm. In: Yin, F.-L., Wang, J., Guo, C. (eds.) ISNN 2004. LNCS, vol. 3173, pp. 211–216. Springer, Heidelberg (2004). doi:10.1007/978-3-540-28647-9_36
4. Chen, Y., Yang, B., Dong, J., Abraham, A.: Time-series forecasting using flexible neural tree model. Inf. Sci. **174**(3), 219–235 (2005)
5. Chen, Z., Peng, L., Gao, C., Yang, B., Chen, Y., Li, J.: Flexible neural trees based early stage identification for IP traffic. Soft Comput. **21**(8), 2035–2046 (2017)
6. Hinton, G.E.: A practical guide to training restricted boltzmann machines. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) Neural Networks: Tricks of the Trade. LNCS, vol. 7700, pp. 599–619. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35289-8_32
7. Le Cun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Handwritten digit recognition with a back-propagation network. In: NIPS 1990, pp. 396–404 (1990)
8. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)