Backpropagation neural tree

Neural Networks 149 (2022) 66–83

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

Backpropagation Neural Tree

Varun Ojha [a,*], Giuseppe Nicosia [b,c]

[a] Department of Computer Science, University of Reading, Reading, UK
[b] Center of System Biology, University of Cambridge, Cambridge, UK
[c] Department of Biomedical & Biotechnological Sciences, University of Catania, Catania, Italy

ARTICLE INFO

Article history:
Received 28 March 2021
Received in revised form 22 August 2021
Accepted 3 February 2022
Available online 10 February 2022

Keywords:
Stochastic gradient descent
RMSprop
Backpropagation
Minimal architecture
Neural networks
Neural trees

ABSTRACT

We propose a novel algorithm called *Backpropagation Neural Tree* (BNeuralT), which is a *stochastic computational dendritic tree*. BNeuralT takes random *repeated inputs* through its leaves and imposes *dendritic nonlinearities* through its internal connections like a biological *dendritic tree* would do. Considering the dendritic-tree like plausible biological properties, BNeuralT is a *single neuron* neural tree model with its internal sub-trees resembling dendritic nonlinearities. BNeuralT algorithm produces an ad hoc *neural tree* which is trained using a stochastic gradient descent optimizer like gradient descent (GD), momentum GD, Nesterov accelerated GD, Adagrad, RMSprop, or Adam. BNeuralT training has two phases, each computed in a depth-first search manner: the *forward pass* computes neural tree's output in a *post-order traversal*, while the error backpropagation during the *backward pass* is performed recursively in a *pre-order traversal*. A BNeuralT model can be considered a *minimal subset* of a neural network (NN), meaning it is a "thinned" NN whose complexity is lower than an ordinary NN. Our algorithm produces high-performing and parsimonious models balancing the complexity with descriptive ability on a wide variety of machine learning problems: classification, regression, and pattern recognition.

1. Introduction

Data-driven learning is a hypothesis (trained model) search from a hypothesis-space that fits input data to its target output as good as possible (a low error on test data). A learning algorithm like neural networks (NNs) parameter optimization via backpropagation is the effort to find such a hypothesis (Rumelhart, Hinton, & Williams, 1986). We propose a new study of ad hoc *neural trees* generation and their optimization via our *recursive backpropagation algorithm* to find such a hypothesis. Hence, we propose a new algorithm called *Backpropagation Neural Tree* (BNeuralT).

A tree of BNeuralT is like a biological dendritic tree (Mel, 2016; Travis, Ford, & Jacobs, 2005) that processes repeated inputs connected to a single neuron (Beniaguev, Segev, & London, 2020; Jones & Kording, 2021) through dendritic nonlinearities (London & Häusser, 2005). Structurally, BNeuralT model is a stochastic *computational dendritic tree* that takes random *repeated inputs* through its leaves and imposes *dendritic nonlinearities* through its internal nodes like a biological dendritic tree would do (Jones & Kording, 2021; Travis et al., 2005). Hence, considering the plausible dendritic-tree-like biological properties, BNeuralT is a *single neuron* neural tree model with its internal nodes resembling dendritic nonlinearities.

Structurally, BNeuralT, being a tree, is a minimal subset of a (highly sparse) NN whose complexity is comparatively low (Poirazi, Brannon, & Mel, 2003b). This means that a NN with a very high *dropout* [a network regularization technique (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014)] prior to its training can be similar to BNeuralT, except BNeuralT has dedicated paths from input to output as opposed to sparse NN that has shared connections between nodes. Hence, we aim to gauge the performance of ad hoc *neural trees* trained using *stochastic gradient descent* (SGD) optimizers like gradient descent (GD), momentum gradient descent (MGD) (Qian, 1999), Nesterov accelerated gradient descent (NAG) (Bengio, Boulanger-Lewandowski, & Pascanu, 2013), adaptive gradient (Adagrad) (Dean et al., 2012), root-mean-square gradient propagation (RMSprop) (Tieleman & Hinton, 2012), and adaptive moment estimation (Adam) (Kingma & Ba, 2015).

Operationally, an expression-tree with its operator (node) being neural nodes (i.e., an operator is an activation function), edges being neural weights, and leaves being inputs make a neural tree architecture, where the tree's architecture itself can be optimized (Chen, Yang, Dong, & Abraham, 2005; Schmidt & Lipson, 2009). The tree's edges (parameters) optimization is straightforward using a gradient-free method (Kennedy & Eberhart, 1995; Rios & Sahinidis, 2013) where the tree is assumed

* Corresponding author.
E-mail address: v.k.ojha@reading.ac.uk (V. Ojha).

https://doi.org/10.1016/j.neunet.2022.02.003
0893-6080/

a target function (Ojha, Abraham, & Snášel, 2017). However, its gradient-based optimization is non-trivial, especially because the error-backpropagation through the tree data structure is recursive to traverse. Our proposed BNeuralT algorithm does a two-phase computation of a neural tree in a depth-first search manner: the *forward pass* computes neural tree's outputs in a *post-order traversal*, while the error backpropagation during the *backward pass* is performed recursively in a *pre-order traversal*.

We trained ad hoc neural trees in an *online mode* (example-by-example) and a *mini-batch mode* on a variety of learning problems: classification, regression, and pattern recognition. For classification and pattern recognition problems, BNeuralT has its root node's children (nodes at tree depth one) strictly dedicated to each target class, and the root node decides the winner class on receiving input data. BNeuralT dedicates its root as the output node for a regression problem.

We evaluated BNeuralT's convergence process on six SGD optimizers and analyzed BNeuralT's complexity against its convergence accuracy. Each training version was compared with a similar training version of a multi-layer perceptron (MLP) algorithm (i.e., an input-hidden-output NN architecture) and classification and regression algorithms such as decision tree (DT) (Breiman, Friedman, Stone, & Olshen, 1984), random forest (RF) (Breiman, 2001), single and multi-objective versions of a heterogeneous flexible neural tree (HFNT^S and HFNT^M) (Ojha et al., 2017), multi-output neural tree (MONT) (Ojha & Nicosia, 2020), Gaussian process (GP) (Rasmussen & Williams, 2006), naïve Bayes classifier (NBC) (Mitchell, 1997), and support vector machine (SVM) (Chang & Lin, 2011; Cortes & Vapnik, 1995; Fan, Chang, Hsieh, Wang, & Lin, 2008). The results on all problems indicate the success of our BNeuralT algorithm that produces high-performing and parsimonious models balancing the complexity and descriptive ability with a minimal training hyperparameters setup.

Our contribution is an innovative Recursive Backpropagation Neural Tree algorithm that

- takes inspiration from biological dendritic trees to solve a wide class of machine learning problems through a single neuron tree-like model performing dendritic nonlinearities through its internal nodes and resembling a highly sparse neural network.
- generates low complexity and high accuracy models. Therefore, we have designed a learning system capable of producing minimal and sustainable neural trees that have fewer parameters to produce more compact and, therefore, sustainable neural models able to reduce CPU time and, consequently, CO_2 emissions for machine learning applications.
- shows that the sigmoidal dendritic nonlinearity of *any* stochastic ad hoc neural tree structure can solve machine learning problems with high accuracy, and any such structure excels to genetically optimized neural tree structures, NNs, and other learning algorithms.

This paper presents relevant related work in Section 2. BNeuralT model's architecture and properties are described in Section 3. Sections 4.1 and 4.2 outline the hyperparameter settings and experiment versions. The performance of BNeuralT on machine learning problems is summarized in Section 5 and discussed in Section 6, followed by conclusions in Section 7. Source code of BNeuralT algorithm and pre-trained models are available at https://github.com/vojha-code/BNeuralT.

2. Related works

We review the works defining neural tree architectures and training processes. The early definition of neural trees appeared in Sakar and Mammone (1993) and Sirat and Nadal (1990), where the tree's *"root-to-leaf"* path is represented as a neural network (NN). Such a tree makes its decision through leaf nodes, and its internal nodes are NNs (or neural nodes). Jordan and Jacobs (1994) proposed a hierarchical mixture expert model that performs construction of a binary tree structure where the model hierarchically combines the outputs of expert networks (feed-forward NNs at the terminal) though getting networks (feed-forward NNs at non-terminal) and propagates computation from *"leaf-to-root"* and where each NN uses the whole input features set.

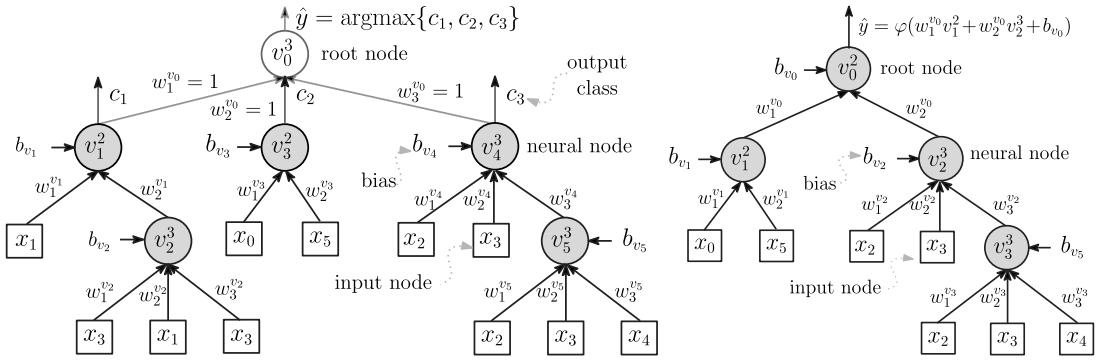
In contrast, our model is purely a single network (tree) structure representation, whereas a hierarchical mixture expert model is a hierarchical combination of several (preferably small) networks. Therefore, unlike the hierarchical mixture expert model, our model is a subset of a NN where "leaf-to-root" has a specific information processing path. In fact, considering plausible inspiration from biological computational dendritic tree (Mel, 2016; Poirazi et al., 2003b; Travis et al., 2005), our model behaves as a single neuron model (Jones & Kording, 2021).

Our proposed BNeuralT algorithm generates an *m-ary* tree structure stochastically and assigns edge weights randomly. BNeuralT's each leaf node (terminal node) takes a single input variable from a set of all available variables (data features). Therefore, in a generated tree, some features could remain unused by the model leading to *only* select features responsible for the prediction. Moreover, tree's each neural node (non-terminal node) takes a weighted summation of its child's output. Hence, a BNeuralT model potentially performs an *input dimension reduction* and propagates the computation from *leaf to root*.

A recent work of Tanno, Arulkumaran, Alexander, Criminisi, and Nori (2019) demonstrates a neural tree as an arrangement of convolution layers and linear classifier as a learning model resembling a decision tree-like classifier where the incoming inputs at the nodes are inferred through the so-called router, processed through tree edges (transformers), and classified through leaf (solver) nodes. In contrast, our model takes image pixels as its inputs. A *leaf-to-root* as a neural tree definition appeared in Chen et al. (2005) and Zhang, Ohm, and Mühlenbein (1997), where the tree's leaf nodes are designated inputs, internal nodes are neural nodes, and edges are weights. Such types of neural trees have been subjected to structure optimization (Chen et al., 2005; Ojha et al., 2017) and parameter optimization via gradient-free optimization techniques like particle swarm optimization (Chen, Yang, & Abraham, 2007) and differential evolution (Ojha et al., 2017).

Zhang et al. (1997) demonstrated that a neural tree could be evolved as a subset of an MLP. Their effort was to evolve a neural tree using genetic programming and optimize parameters using a genetic algorithm. Lee, Gallagher, and Tu (2016) focused on implementing pooling layers within a convolutional NN as a tree structure. However, our approach is to generate and train ad hoc neural trees using our proposed recursive backpropagation algorithm. To the best of our knowledge and review, this is the first and novel attempt to generate and train *ad hoc* neural trees using our *recursive error-backpropagation* algorithm. Our motivation is to avoid any prior assumptions on network architecture and complicated hyperparameter settings.

Srivastava et al. (2014) proposed a dropout technique that suggests randomly dropping neurons from a large NN. This creates "thinned" NN instances during training and prevents a NN from overfitting. Our proposed BNeuralT randomly generates a tree architecture, which can be considered a sparse NN in a similar sense with rather a higher dropout. Also, the branching and pruning of the tree branches in BNeuralT are performed at the tree generation stage, where a branch is probabilistically pruned by generating a leaf node at a depth lower than terminals.



(a) Classification neural tree (b) Regression neural tree

Fig. 1. Neural trees: (a) A neural tree example of a three-class classification learning problem. The root node v_0^3 takes three immediate children: v_1, v_3, and v_4, each respectively designated to a class c_1, c_2, and c_3. The internal nodes (shaded in gray) are neural nodes and take an activation function $\varphi(\cdot)$ and leaf nodes are inputs. Each designated output class has its subtree. This tree takes its input from the set $\{x_1, x_2, \ldots, x_5\}$. The link $w_i^{v_j}$ between nodes are neural weights. (b) A neural tree example for a regression problem has one output node v_0.

3. Backpropagation neural tree

3.1. Problem statement

Let $\mathcal{X} \in \mathbb{R}^d$ be an instance-space and $\mathcal{Y} = \{c_1, \ldots, c_r\}$ be a set of r labels such that a label $y \in \mathcal{Y}$ is assigned to an instance $\mathbf{x} \in \mathcal{X}$. Therefore, for a training set of N instance-label pairs $\mathcal{S} = (\mathbf{x}_i, y_i)_{i=1}^N$, we induce a classifier $\mathcal{G}(\mathcal{X}, \mathbf{w})$ that reduces classification cost $\mathcal{L}_{\text{Error}}(\mathcal{G}) = {}^1\!/_N \sum_{i=1}^N (\hat{y}_i \neq y_i)$, where $\hat{y}_i$ is a predicted output class on an input instance $\mathbf{x}_i = \langle x_1^i, x_2^i, \ldots, x_d^i \rangle$ labeled with the target class $y_i \in \{c_1, \ldots, c_r\}$. Additionally, when an instance $\mathbf{x} \in \mathcal{X}$ is associated with a continuous variable $y \in \mathbb{R}$ rather than a set of discrete class labels, then $\mathcal{G}(\mathcal{X}, \mathbf{w})$ is a predictor that for a training set of instance-output pairs $\mathcal{S}$ reduces a prediction cost like mean squared error (MSE) $\mathcal{L}_{\text{MSE}}(\mathcal{G}) = {}^1\!/_N \sum_{i=1}^N (\hat{y} - y_i)^2$.

3.2. Backpropagation neural tree algorithm

Backpropagation neural tree (BNeuralT) takes a tree-like architecture whose root node is a decision node, and leaf nodes are inputs. For classification learning problems, BNeuralT has strictly dedicated nodes at level-1 (child nodes of the root) of the tree to represent classes. Fig. 1(a) is an example of a *classification neural tree* where root's each immediate child is a sub-tree dedicated to a class, and the root node only decides a winner class $\hat{y} = \text{argmax}\{c_1, \ldots, c_r\}$ for an instance-label pair $(\mathbf{x}, y)$. For regression learning problems, BNeuralT is a *regression neural tree* whose root node decides the tree's predicted output $\hat{y} = \varphi(\sum_{i=1}^{child} w_i v_i + b_{v_0})$, where $\varphi(\cdot)$ is an activation function yielding a value in $[0, 1]$, w_i is a edge weight, v_i is the activation of ith child, and b_{v_0} is the root's bias (cf. Fig. 1(b)).

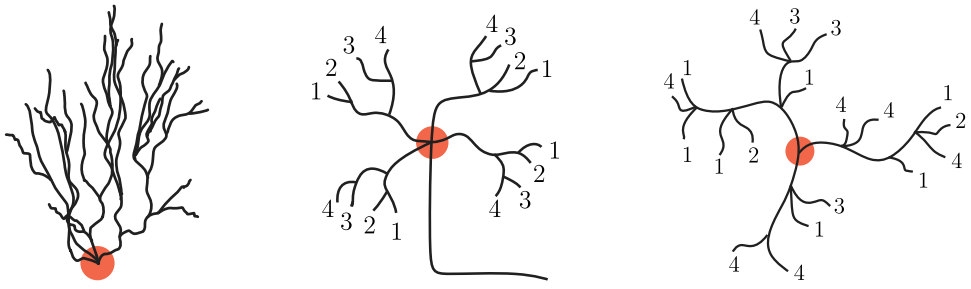
BNeuralT, denoted as $\mathcal{G}$ is an *m–ary* rooted tree with its one node designated as the root node, and each node takes at least $m \geq 2$ child nodes except for a leaf node that takes no child node. Hence, for a tree depth p, BNeuralT takes $n \leq [(m^{p+1} - 1)/(m - 1)]$ nodes (including the number of internal nodes $|V|$ and the leaf nodes $|T|$). Thus, BNeuralT can be defined as a union of internal and leaf nodes $\mathcal{G} = V \cup T = \left\{v_1^j, v_2^j, \ldots, v_K^j\right\} \cup \{t_1, t_2, \ldots, t_L\}$ where kth node $v_k^j \in V$ is an internal node and receives $2 \leq j \leq m$ inputs from its child nodes. The kth leaf node $t_k \in T$ has no child, and it has a designated input $x_i \in \{x_1, x_2, \ldots, x_d\}$.

Fig. 1 is an example of classification (left) and regression (right) trees. All internal nodes (shaded in gray) of the tree are neural nodes and may behave like the nodes of a NN. That is, a neural node computes a weighted summation z of inputs and squashes that using an activation function $\varphi(z)$, e.g., *sigmoid*: $\varphi(z) = {}^1\!/_{(1+e^z)}$ or *ReLU*: $\varphi(z) = \max(0, z)$. We installed sigmoid or ReLU functions as BNeuralT's neural nodes, which can be any other activation function like *tanh*. The trainable parameters $\mathbf{w}$ are the edges and the bias weights of the nodes. The number of nodes n in a tree grows as per $\mathcal{O}(m^p)$. The number of edges $(n - 1)$ is proportional to the growth of n, so is the number of tree's *trainable parameters* $\mathbf{w}$.

Complexity of BNeuralT. A BNeuralT model resembles an expression tree, and its computation is a depth-first-search post-order or pre-order traversal where each node needs to be visited at least once. Hence, the worst-case *time complexity* of BNeuralT is $\mathcal{O}(n)$, n being the number of nodes. In a BNeuralT model, each internal node has a bias, each leaf has an input, and each edge has a weight. Therefore, the space requirement of a BNeuralT model is $2|V|+|T|$, i.e., two times internal nodes plus leaf nodes, which will grow proportional to the growth of tree's total nodes $n = |V|+|T|$. Thus, BNeuralT's worst-case *space complexity* is $\mathcal{O}(n)$.

Biologically plausible neural computation of BNeuralT. A typical NN uses neurons inspired by the work of McCulloch and Pitts (1943). Such a neuron operates on a weighted sum of inputs and processes the sum via a nonlinear threshold function. Such neural computation considers that the dendrites (synaptic inputs) of a neuron are summed at "soma", thereby exciting a neuron, i.e., providing it a firing strength (Hodgkin & Huxley, 1952; McCulloch & Pitts, 1943; Poirazi, Brannon, & Mel, 2003a). However, the biological behavior of dendrites shows that dendrites themselves impose nonlinearity on their synaptic inputs before summing at "soma" (Hay, Hill, Schürmann, Markram, & Segev, 2011; London & Häusser, 2005). This dendritic nonlinearity is possibly a sigmoidal nonlinearity (Poirazi et al., 2003b). Additionally, the synaptic connections in a fully connected NN are symmetric, whereas biological dendritic connections are asymmetric (Farhoodi & Kording, 2018; Mel, 2016; Travis et al., 2005) [cf. Fig. 2(a)].

Poirazi et al. (2003b) using a sparse two-layer NN analogous to a binary tree-like dendritic NN has shown the possibility of modeling a single neuron as a NN. The work of Beniaguev et al. (2020) demonstrated a proof of concept single neuron model as a synaptic *integration and fire* model capable of performing classification of two types of classes with a high degree of temporal accuracy. Jones and Kording (2021) considered the biologically asymmetric morphology of "*dendritic tree*" and its repeated synaptic inputs to a neuron to show the computational capability of a *single neuron* for solving machine learning problems.



(a) Travis et al. (2005) (b) Jones and Kording (2021) (c) BNeuralT

Fig. 2. Biologically plausible neural computation using *dendritic trees*. Red circle represents a neuron (*soma*), black lines are dendrites, and the numbers indicate inputs.

Fig. 2(b) shows Jones and Kording (2021)'s a single neuron computational model with repeated inputs as a binary tree structure. Unlike the work of Poirazi et al. (2003b), BNeuralT has asymmetric *dendritic connections* to a *"single" neuron* (Beniaguev et al., 2020; Jones & Kording, 2021). Jones and Kording (2021)'s dendritic tree has a systematic and regular binary-tree-like structure and solves a binary classification problem. Whereas BNeuralT's neuron is like the neuron of Travis et al. (2005) (cf. Fig. 2(a)), and it has a stochastic m-ary rooted tree-like structure (cf. Fig. 2(c)). Thus, Through BNeuralT, we investigate the ability of a single neuron with sigmoidal nonlinearity (and linear when using ReLU) in its dendritic connections on three machine-learning problems: multi-class classification, regression, and pattern recognition.

Stochastic gradient descent (SGD) training. BNeuralT's trainable parameters $\mathbf{w}$ are iteratively optimized by a stochastic gradient descent (SGD) method (cf. Algorithm 1) that at an iteration j requires gradient $\nabla\mathbf{w}_j \leftarrow \textsc{Gradient}\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})$ computation (cf. Algorithm 2) and weight update as per $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} + \eta\nabla\mathbf{w}_j$. The weight update $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} + \eta\nabla\mathbf{w}_j$ in line number 7 of Algorithm 1 is a simple GD method, where other similar optimizers like MGD, NAG, Adagrad, RMSprop, or Adam can also be used. Table 1 details the expressions of weight updates for these optimizers.

Algorithm 1 Stochastic gradient descent (SGD)

1: **procedure** SGD($\mathcal{G}_{\mathbf{w}}$, $\mathcal{S}$) ▷ SGD $(\cdot,\cdot)$ takes a neural tree $\mathcal{G}_{\mathbf{w}}$ and training data $\mathcal{S}$
2: $\mathbf{w}_0 \leftarrow \mathcal{G}_{\mathbf{w}}$ ▷ initial weights
3: **for** epoch$_i$ < epoch$_{\max}$ **do**
4: $\mathcal{S} \leftarrow \textsc{Shuffle}(\mathcal{S})$ ▷ function Shuffle returns a randomly shuffle dataset
5: **for** an instance $\mathbf{x}_j \in \mathcal{S}$ **do**
6: $\Delta\mathbf{w}_j \leftarrow \textsc{Gradient}\ \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})$ ▷ computed as per Algorithm 2
7: $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} + \eta\Delta\mathbf{w}_j$ ▷ GD weights update on an input instance $\mathbf{x}_j$
8: **end for**
9: **end for**
10: **end procedure**

Error-backpropagation in BNeuralT. Our proposed *recursive error-backpropagation* in BNeuralT algorithm has two computation phases: *forward pass* and *backward pass* (cf. Fig. 3). Both work in a *depth-first search* manner. Since a tree data structure is algorithmically recursive to traverse through, both forward pass and backward (error-backpropagation) pass take place in a recursive manner. The forward pass computation produces the output for a tree in a *post-order traversal* manner (cf. Fig. 3(left)). That is, each leaf node propagates its input through dendrite (edge) to its parent node, and subsequently, each internal node, after computing received inputs from its child nodes, propagates activation to their respective parent node. Finally, the root node computes the tree's output.

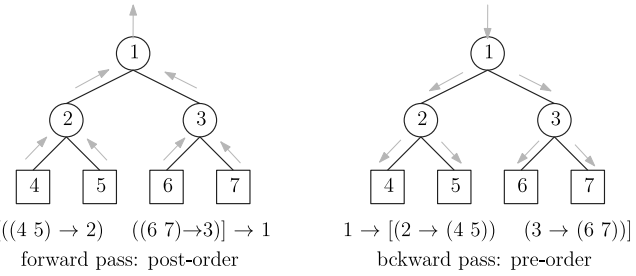


Fig. 3. Forward pass (left) and backward pass (right) computation. The arrows show the direction of computation.

The backward pass computes the gradient of the error with respect to edge weights. The backward pass computes gradient δ for each internal node and propagates it back to each edge depth-by-depth. Hence, the backward pass is a *pre-order traversal* of the tree (cf. Fig. 3(right)). That is gradient δ computed at the root node flow backward to its child node until it reaches leaf nodes. Fig. 4 (left) shows the forward pass and backward pass computation labeled with variables of neural tree computation. Fig. 4 (right) shows the backpropagation of gradient from an output node to the inputs (leaf nodes). Algorithm 2 is a summary of error-backpropagation and Algorithm 3 shows recursive gradient computation for BNeuralT that facilitates error-backpropagation.

4. Experiments

4.1. Hyperparameters settings

Datasets. We select a set of nine *classification problems*: Australia (Aus), Heart (Hrt), Ionosphere (Ion), Pima (Pma), Wisconsin (Wis), Iris (Irs), Wine (Win), Vehicle (Vhl), Glass (Gls), which respectively have 14, 13, 33, 8, 30, 4, 13, 18, and 9 input attributes; 2, 2, 2, 2, 2, 3, 3, 4, and 7 target classes; and 690, 270, 351, 768, 569, 150, 178, 846, and 214 examples. For *regression problems*, we select Baseball (Bas), Daily Electricity Energy (Dee), Diabetes (Dia), Friedman (Frd), and Miles Per Gallon (Mpg), which respectively have 16, 6, 10, 5, and 6 input attributes and 337, 365, 442, 1200, and 392 examples. Each regression dataset has one target. These datasets are available at Bache and Lichman (2013) and Keel (2011). These problems are significantly different not only in terms of the number of classes and examples but also in terms of their attribute types and range. This differing nature of these problems poses significant variations in difficulty for one algorithm to excel on all problems (Wolpert, 1996).

Table 1
Gradient descent versions to replace line number 7 in Algorithm 1. Symbols η, γ, β, β_1, and β_2 are constants (hyperparameter) of respective algorithms. Symbols v_j and w_j show previous momentum and weights, respectively, and $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})$ shows gradient of loss $\mathcal{L}$ over input $\mathbf{x}_j$ and $\mathbf{w}$ of tree $\mathcal{G}$.

Algorithm	Expression
MGD (Qian, 1999)	$v_j \leftarrow \gamma v_{j-1} + \eta\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})$ $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} - v_j$
Nesterov accelerated GD (Bengio et al., 2013)	$v_j \leftarrow \gamma v_{j-1} + \eta\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j - \mathbf{w}_{j-1}})$ $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} - v_j$
Adagrad (Dean et al., 2012)	$v_j \leftarrow v_{j-1} + \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})^2$ $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} - \eta/\sqrt{v_j+\epsilon}\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})$
RMSprop (Tieleman & Hinton, 2012)	$v_j \leftarrow (1-\gamma)v_{j-1} + \gamma\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})^2$ $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} - \left(\eta/\sqrt{v_j+\epsilon}\right)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})$
Adam (Kingma & Ba, 2015)	$m_j \leftarrow \beta_1 m_{j-1} + (1-\beta_1)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})/\left(1-\beta_1^j\right)$ $v_j \leftarrow \beta_2 v_{j-1} + (1-\beta_2)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}_j, \mathcal{G}_{\mathbf{w}_j})^2/\left(1-\beta_2^j\right)$ $\mathbf{w}_j \leftarrow \mathbf{w}_{j-1} - \eta/\left(\sqrt{v_j+\epsilon}\right)m_j$

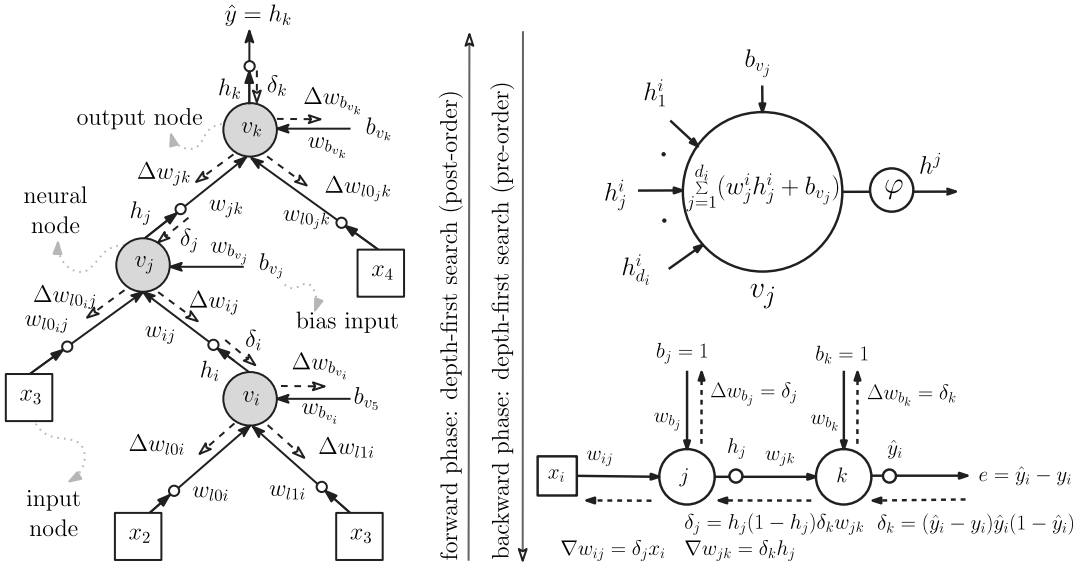


Fig. 4. *Left.* Backpropagation neural tree. Output node v_k yields output y using forward pass upon receiving inputs x_i from leaf nodes. Each node is linked with an edge weight w_{ij}. The backward pass propagates error $e = (y - \hat{y})$ back to input nodes to compute weight change Δw. *Right.* Backpropagation of error from an output node, k; to a hidden node, j; to an input node, i; and to bias inputs b_k and b_j. Dashed lines represent error backpropagation and computation δ and gradient ∇w (cf. Algorithm 2) to find weight change Δw that help stochastic gradient descent (cf. Algorithm 1).

Both classification and regression learning datasets were *normalized* using min–max normalization between 0 and 1. Each dataset was randomly shuffled and partitioned into training (80%) and test (20%) sets for each instance of the experiment. For a *pattern recognition* problem, we select the MNIST dataset (LeCun, Cortes, & Burges, 2020), which has 60,000 training examples and 10,000 test examples labeled with a set of 10 handwritten characters, and this dataset was normalized by dividing gray-scale pixel value by 255.

BNeuralT hyperparameters. We repeated experiments 30 times (independently) for each classification and regression problem. In each run, we generated ad hoc BNeuralTs (stochastically generated tree structures) for each dataset with a maximum tree depth $p = 5$; max child per node $m = 5$, and branch pruning factor $P[\text{leaf}_p < p] \in \{0.4, 0.5\}$ which is a probability of a leaf node being generated at a depth lower than the tree height p. A higher leaf generation probability (e.g., 0.5) at internal nodes means that tree height terminates earlier than its predefined depth, which means a tree will be generated with fewer parameters. A lower leaf generation probability (e.g., 0.4) means a deeper tree structure with more parameters.

For classification problems, BNeuralT's output neural node was *argmax* node (i.e., winner takes all node), whereas, for regression problems, it was *sigmoid* activation function. Its internal nodes were *sigmoid* activation (or *ReLU* for some trial experiment versions). For pattern recognition (MNIST dataset), BNeuralT models were generated on an ad hoc basis via setting maximum tree depth $p \in \{5, 10\}$, and max child per node $m \in \{10, 15\}$ with $P[\text{leaf}_p < p] \in \{0.4, 0.5\}$ and a tree size threshold $|\mathcal{G}|_{\geq \text{MIN}} \in \{1\text{K}, 20\text{K}\}$, where "K" is 1000.

Other algorithms hyperparameters. MLP architecture was a fixed three-layer architecture [inputs–hidden (100 nodes) –targets] for each dataset of classification and regression problems. A *SoftMax* layer acted as the MLP classifier's output nodes, and an MLP regression had a *sigmoid* activation as its output node. The internal neural nodes in an MLP were *sigmoid* (or *ReLU*) functions. Other algorithms HFNT$^{\text{S}}$, HFNT$^{\text{M}}$, MONT$_3$, DT, RF, GP, NBC, SVM, and CARTs had their default setups as they are in their

Algorithm 2 Backpropagation computation of neural tree $\mathcal{G}$

procedure GRADIENT $\nabla\mathbf{w}\mathcal{L}(\mathbf{x}, \mathcal{G})$▷ $\mathbf{x}$ and $\mathbf{w}$ are the inputs and trainable parameters
  $\mathcal{G}_\delta \leftarrow$COMPUTE δ $(y, \mathcal{G}(\mathbf{x}), N_0 \leftarrow \mathcal{G}, \mathcal{G})$ ▷ Compute δ using Algorithm 3 for target y and prediction $\mathcal{G}(\mathbf{x})$ at tree's root N_0
  for all nodes N in $\mathcal{G}_\delta$ **do**
    if $N \rightarrow Type \rightarrow$ Output node **then** ▷ output node's bias weight update
      $N \rightarrow \nabla w_{b_k} = N \rightarrow \delta_k$ ▷ bias weight is set to current node's gradient δ
    else if $N \rightarrow Type \rightarrow$ Internal (hidden) node **then**
      $h_j = N \rightarrow h_j$ ▷ current node's activation is the incoming signal for weight w_{jk}
      $\delta_k = N \rightarrow N_{Parent} \rightarrow \delta$ ▷ gradient back-propagated from the parent node
      $N \rightarrow \nabla w_{jk} = \delta_k h_j$▷ w_{jk} is the weight between node N_j and its parent node N_k
      $N \rightarrow \nabla w_{b_j} = N \rightarrow \delta_j$▷ bias weight is set to current node's gradient δ
    else if $N \rightarrow Type \rightarrow$ Leaf node **then**
      $x_i = N \rightarrow x$ ▷ $N \rightarrow x \in \{x_1, x_2, \ldots, x_d\}$ is an input attribute
      $\delta_j = N \rightarrow N_{Parent} \rightarrow \delta$ ▷ gradient back-propagated from parent to child
      $N \rightarrow \nabla w_{ij} = \delta_j x_i$ ▷ w_{ij} is the weight between child N_i and parent N_j
    end if
  end for
return $\nabla\mathbf{w}$
end procedure

Algorithm 3 Computation of δ for each neural node

procedure COMPUTE δ $(y, \hat{y}, N \leftarrow \mathcal{G}, \mathcal{G})$ ▷ y is a target, $\hat{y}$ is a prediction, and N is the current (or entry) node of tree $\mathcal{G}$
  if $N \rightarrow Type \rightarrow$ Output node **then** ▷ compute gradient δ_k for output node N in $\mathcal{G}$
    $N \rightarrow \delta_k = (\hat{y} - y)\hat{y}(1 - \hat{y})$ ▷ gradient at sigmoid output node
  else ▷ compute gradient at an internal (hidden) neural node
    $h_j = N \rightarrow h_j$ ▷ activation (value) of an internal (neural) node
    $\delta_k = N \rightarrow N_{Parent} \rightarrow \delta_k$ ▷ retrieve parent's gradient δ of the current node N
    $w_{jk} = N \rightarrow w_{jk}$ ▷ edge (weight) between current node N_j and parent node N_k
    $N \rightarrow \delta_j = h_j(1 - h_j)\delta_k w_{jk}$ ▷ gradient at an internal sigmoid node
  end if
  for all child (neural) node N_c of N in $\mathcal{G}$ **do**
    $\mathcal{G}_\delta \leftarrow$COMPUTE δ $(\emptyset, \emptyset, N_c \leftarrow \mathcal{G}, \mathcal{G})$ ▷ call COMPUTE δ, $\emptyset$ indicates unused argument
  end for
end procedure

libraries (Pedregosa et al., 2011) or in the literature (Ojha et al., 2017; Ojha & Nicosia, 2020; Zharmagambetov, Hada, Carreira-Perpiñán, & Gabidolla, 2019). A detailed list of hyperparameters of all algorithms is provided in Supplementary Table A1.

SGD hyperparameters. BNeuralT and MLP algorithms take *optimizers* like GD, MGD, NAG, Adagrad, RMSprop, or Adam. The training parameters were learning rate $\eta = 0.1$, momentum rate $\gamma = 0.9$, $\beta_1 = 0.9$, $\beta_2 = 0.9$, $\epsilon = 1e^{-8}$, training mode was *stochastic* (online), and training epochs were 500. Since the gradient descent computation was stochastic, both BNeuralT and MLP do the same number of forward-pass (function) evaluations, i.e., number training examples $\times$ epochs. All six optimizers were used for training BNeuralT and MLP with an *early-stopping* restore-best strategy (or without an *early-stopping* for some trail experiments), whereas other algorithms take their own default optimizer (Pedregosa et al., 2011). While BNeuralT and MLP were trained in *online* mode (example-by-example training), other algorithms take only *offline* mode (epoch-by-epoch) training.

For the pattern recognition problem (MNIST), we set a *mini-batch* training with a batch size of 128 examples. RMSprop was used as an *optimizer*, and BNeuralT was trained by varying learning rate $\eta \in \{0.1, 0.01\}$ and the number of epochs $\in \{10, 25, 50, 70\}$. The results of other algorithms on MNIST were collected from literature to compare performances.

Loss functions. The loss function for BNeuralT training for classification and pattern recognition problems was a miss-classification rate $\mathcal{L}_{\text{Error}}(\mathcal{G})$. MLP training on classification problems was best with categorical cross-entropy loss (Bishop, 2006). The training of other algorithms had default setups recommended in their libraries (Pedregosa et al., 2011). For regression problems, all algorithms were trained by reducing $\mathcal{L}_{\text{MSE}}(\cdot)$. The test metric for classification problems for all algorithms was a miss-classification rate $\mathcal{L}_{\text{Error}}(\cdot)$ and for regression problems, it was a regression fit $\mathcal{L}_{r^2}(\cdot) = 1 - \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 / \sum_{i=1}^{N}(y_i - \bar{y})^2$ (Nash–Sutcliffe model efficiency coefficient) which gives a value between $[-\infty, 1]$, where $\bar{y}$ is the mean of target y.

Forward pass computation time, τ. BNeuralT was implemented in Java 11, and MLP was implemented using TensorFlow and Keras libraries (Keras, 2020). Other algorithms were implemented using scikit-learn library (Pedregosa et al., 2011) in Python 3.5. The forward pass computation time, τ is a wall-clock time on Windows 10 operating system with configuration x64 Intel i5-2400 CPU, 3.1 GHz, 3101 MHz, 4 Cores, and 16 GB physical memory.

4.2. BNeuralT and MLP experiment versions

We experimented with multiple versions of BNeuralT and MLP settings to bring out the best of both. We tried *sigmoid* and *ReLU* as the internal activation functions. We tried BNeuralT's branch pruning factor $P[\text{leaf}_p < p]$ with 0.5 and 0.4.

The learning rates of the optimizers had two sets: (i) A flat learning rate $\eta = 0.1$ for all optimizers. (ii) The learning rate recommended in the Keras library for respective optimizers, i.e., RMSprop, Adam, and Adagrad had $\eta = 0.001$, and MGD, NAG, and GD had $\eta = 0.01$. We call the library's recommended η value a *default* learning rate. In addition, the SGD learning was tried "with" and "without" *early-stopping* (ES) strategies.

These variations produced five BNeuralT settings: (i) ES training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.5 leaf generation rate; (ii) ES training of BNeuralT having *sigmoid* nodes, *default* learning rate, and 0.5 leaf generation rate; (iii) ES training of BNeuralT having *ReLU* nodes, 0.1 learning rate, and 0.5 leaf generation rate; (iv) ES training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate; and (v) without ES training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.5 leaf generation rate.

Multiple MLP settings were tried. Out of which, some best performing settings were: (i) ES training of MLP having *sigmoid* nodes and 0.1 learning rate; (ii) ES training of MLP having *sigmoid* nodes and *default* learning rate; (iii) ES training of MLP having *sigmoid* nodes, 0.1 learning rate, and L2-norm regularization; (iv)

ES training of MLP having *sigmoid* nodes, *default* learning rate, and L2-norm regularization; and (v) experiment setting same as (i) but without ES; and (vi) experiment setting same as (ii) but without ES. Other trials were using dropout with and without early stopping.

For each algorithm (BNeuralT, MLP, $HFNT^S$, $HFNT^M$, $MONT_3$, DT, RF GP, NBC, and SVM), each optimizer (GD, MGD, NAG, Adagrad, RMSprop, and Adam), and each variation of hyperparameter setting, there were 110 experiments (cf. Tables A2 and A3 in Supplementary). We repeated each experiment for each dataset for 30 independent runs, and their average performance on test sets was evaluated.

5. BNeuralT performance

5.1. Selection of the best performing setting

We selected the best performing setting based on the average test accuracy computed over 30 independent runs of BNeuralT, MLP, $HFNT^S$, $HFNT^M$, $MONT_3$, DT, RF, GP, NBC, and SVM to report them in detail in this section. The best performing BNeuralT setting was the "ES training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate". The best MLP setting was "ES training of MLP having *sigmoid* nodes and *default* learning rate". We found that $HFNT^S$, $HFNT^M$, DT, RF, GP, NBC, and SVM worked best with their recommended setting.

We found that collectively on all classification and regression datasets, BNeuralT with *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate trained using RMSprop performed the best among all experiment versions of all algorithms. This setting produced an average accuracy of 83.2% across all datasets with an average of 222 trainable parameters. This same setting also performed the best across all classification datasets among all algorithms, i.e., it produced an average accuracy of 89.1% with an average of 261 trainable parameters. In fact, the top six best results over classification datasets were from BNeuralT settings. GP algorithm came 7th with an average classification accuracy of 86.79%. MLP with *sigmoid* node and ES training using MGD optimizer with *default* learning performed 8th with an average accuracy of 86.78% with an average 1970 trainable parameters.

MLP, however, performed slightly better on regression problems than the other algorithms. MLP with *sigmoid* node and ES training using NAG optimizer with *default* learning rate produced an average regression fit value of 0.775. Whereas BNeuralT with *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate trained using RMSprop optimizer produced an average regression fit value of 0.727. It is important to note that this performance of BNeuralT comes with a much lower average trainable parameter. BNeuralT used only 152 trainable parameters compared to MLP that used 1041 parameters. This means BNeuralT's performance comes with an order magnitude less parameter than MLP on both classification and regression tasks.

Table 2 reports the details of each algorithm's best-performing settings. However, exhaustive lists of 110 experiments, from which we selected these best performing settings, are provided in Supplementary Tables A2 and A3.

5.2. BNeuralT models summary

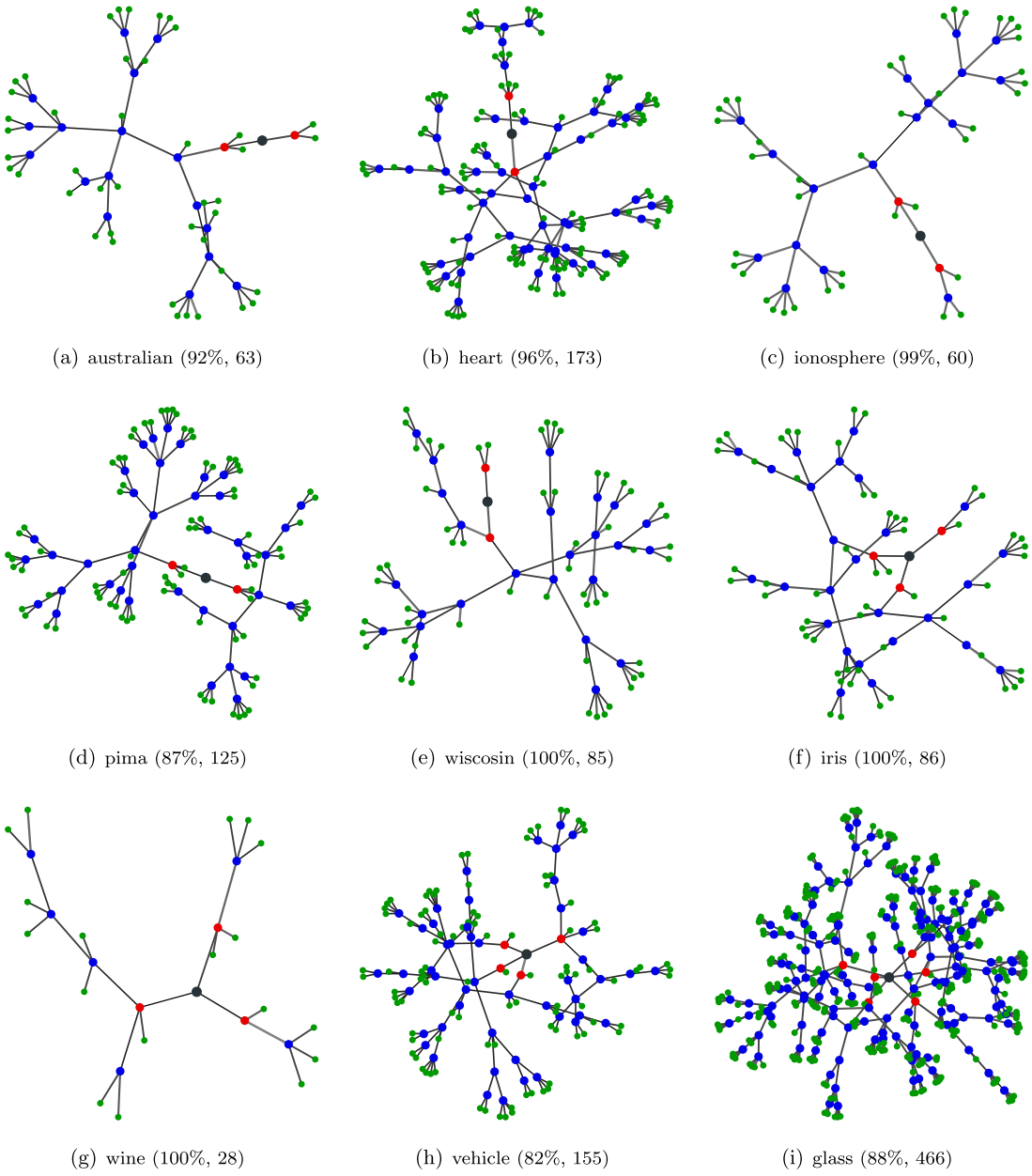
BNeuralT classification models summary. Table 2 suggests that BNeuralT performance on both classification and regression problems is highly competitive with MLP and other algorithms. For example, the average performance of BNeuralT's RMSprop on all classification problems is 2.65% (average accuracy: 89.1%) higher than the nearest best performing non-BNeuralT algorithm. The best MLP model offered an average accuracy of 86.8%, and MLP with a 0.4 dropout rate using Adam produced an 85.9% accuracy. Other algorithms were as follows: $HFNT^S$, 78.9%; $HFNT^M$, 72.4%; $MONT_3$, 83.1%; DT, 81.3%; RF, 86.4%; GP, 86.8%; NBC, 78.2%; and SVM, 84.1%. For this performance, BNeuralT uses *only* 13.25% ($\mathbf{w} = 261$) trainable parameters than MLP's 1969 parameters. The structures of some select best performing BNeuralT classification models are shown in Fig. 5, where black edges indicate dendrites; and green, blue, red, and black nodes, respectively indicate inputs, dendritic nonlinearities, root, and class nodes.

The average tree size of BNeuralT with $P[\text{leaf}_p < p] = 0.5$ and RMSprop was 119 (89% accuracy). The average tree size of $HFNT^M$, $MONT_3$, and $HFNT^S$ algorithms were 29 (72.4% accuracy), 36 (83.1% accuracy), and 92 (78.9% accuracy), respectively. Since tree construction and forward pass computation are similar for BNeuralT, HFNT, and MONT algorithms, there is a trade-off between the model's compactness and accuracy. In fact, this produces a set of trade-off solutions (between accuracy and complexity). Along with this set of Pareto solutions, one can choose which is the best candidate solution for the given machine learning problem under examination: more accurate but less sustainable or a little less accurate but more robust and sustainable.

The forward pass computation time on a single example (in multiple of 10^{-6} s) of BNeuralT was 11.2 s, whereas MLP took 1288; DT, 3.1; RF, 485.4; GP, 455.6; NBC, 31.9; and SVM, 16.1 s. DT was the fastest, and BNeuralT was the second-fastest. However, DT has a much lower accuracy (81.3%) than BNeuralT (89.1%). The time computation is difficult to compare as the algorithms were implemented in different programming languages (Pereira et al., 2017). BNeuralT was implemented in Java 11, and all other algorithms were implemented in Python 3.5. However, BNeuralT's performance on classification problems was clearly better among all algorithms. This is further evident from BNeuralT's collective average accuracy of all optimizers on all classification datasets was 86.1%. Whereas on all optimizers, MLP's average accuracy was 83.8%, tree algorithms had 80.4%, and other algorithms had 83.6% accuracy.

We selected BNeuralT's best optimizer RMSprop for the statistical significance test. This test was designed to examine whether the performance of BNeuralT's RMSprop is statistically significant than that of the other algorithms. Table 3 presents Kolmogorov–Smirnov (KS) test results on two samples to examine the *null hypothesis* that there is no difference between the performance distributions of BNeuralT's RMSprop and other algorithms. The results show that for most datasets and most algorithms, the classification results of BNeuralT's RMSprop show statistical significance over other algorithms' performance as the *null hypothesis* of no difference is rejected in most cases. This was the case, despite using a restrictive Bonferroni correction to adjust p-values. Wilcoxon signed-rank test and Independent T-test in Supplementary Table A4 and Table A5 also favor BNeuralT's RMSprop.

BNeuralT regression models summary. MLP's Adam performed best for regression problems. MLP's Adam produced an average regression fit of 0.772 without dropout and 0.754 with dropout on all datasets. BNeuralT's RMSprop offered an average regression fit of 0.727, which differs only by 5.8% with the best MLP result. This performance of BNeuralT comes with the use of only 14.6% trainable parameters than the parameters used by the MLP ($\mathbf{w} = 1014$). (Note that an MLP dropout model during its test phase uses all weights since dropout only regularizes weights by averaging gradient over epochs during the training phase (Srivastava et al., 2014).) This suggests that BNeuralT is highly capable of learning data with very low complexity with a faster forward pass computation time. The structure of some select best performing BNeuralT regression models is shown in Fig. 6.



(a) australian (92%, 63) (b) heart (96%, 173) (c) ionosphere (99%, 60)

(d) pima (87%, 125) (e) wiscosin (100%, 85) (f) iris (100%, 86)

(g) wine (100%, 28) (h) vehicle (82%, 155) (i) glass (88%, 466)

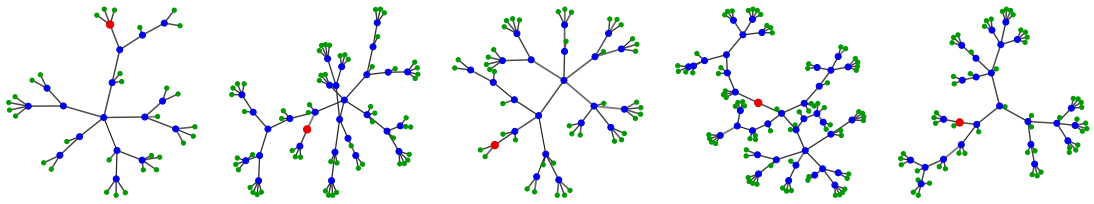
Fig. 5. Classification trees. (a)–(i) show test accuracy and tree size of select high performing trees of datasets. The black node in a tree is its root node, class output nodes are in red, function nodes are in blue, and leaf nodes are in green. The link connecting nodes are neural weights. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The average tree size of BNeuralT with $P[\text{leaf}_p < p] = 0.5$ and RMSprop for regression problems was 64 ($\mathcal{L}_{r^2} = 0.675$). The average tree size of HFNT$^{\text{S}}$ and HFNT$^{\text{M}}$ algorithms were 127 ($\mathcal{L}_{r^2} = 0.562$) and 90 ($\mathcal{L}_{r^2} = 0.567$) respectively. Here, BNeuralT was able to perform accurately compared to genetically optimized HFNT algorithms with less complex models.

The statistical tests in Table 3 also suggest that BNeuralT's RMSprop performance distribution on regression problems compared with MLP's Adam is statistically insignificant *only* on the Friedman dataset. On all other regression datasets, BNeuralT's RMSprop performance is equally significant as other algorithms.

BNeuralT pattern recognition (MNIST) models summary. RMSprop optimizer was found robust and converging fastest for classification models (cf. Section 5.3). Hence, we train BNeuralT on the MNIST character classification dataset (LeCun et al., 2020) using RMSprop. We fed BNeuralT with pixels of MNIST character images since we do not use convolution in BNeuralT. We aimed at generating varied BNeuralT models with varied trainable parameters length by varying tree size. We hoped that a low complexity (few parameters) BNeuralT model would perform competitively with a few reported state-of-the-art. Therefore, we compared BNeuralT performance to gauge its robustness not only on learning small-scale problems reported in Table 2 but on large-scale learning problems like MNIST. Table 4 summarizes BNeuralT models compared with the performances of tree-based state-of-the-art classification algorithms.

Table 4 presents MNIST (pixels) results compared with classification trees. BNeuralT performs the best among the reported trees that work on MNIST (pixels) for character classification. However, convolution of images has been proven efficient for image classification problems. For example, CapsNet (Sabour, Frosst, & Hinton, 2017), a state-of-the-art algorithm on MNIST (convolution), has an error rate of 0.25, but it uses 8 *million parameters*.



(a) baseball (.85, 48) (b) dee (.89, 89) (c) diabetese (.63, 67) (d) friedman (.95, 116) (e) mpg6 (.9, 82)

Fig. 6. Regression trees. (a)–(e) show best performing tree structure of the respective dataset, their accuracy and tree size shown in brackets. The red node in a tree is the root node (output node), function nodes are blue, and leaf nodes are green. The link connecting nodes are neural weights. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
BNeuralT and other algorithms performance as per average (avg.) accuracy $(1 - \mathcal{L}_{\text{Error}}(\cdot))$ and avg. regression fit $(\mathcal{L}_{r^2}(\cdot))$ on the test sets of 30 independent runs for nine classification and five regression learning problems. Both accuracy and regression fit take 1.0 as the best value. Trainable parameters ($\mathbf{w}$) of BNeuralT and MLP are neural weights. The best accuracy among all algorithms is marked in bold. Both BNeuralT and MLP are trained on an online mode, whereas other algorithms take offline mode training. Average forward pass (single example prediction time) wall-clack time is $\tau \times e^{-6}$ s (a lower value is better), where τ^J and τ^P respectively indicate time for Java 11 and Python 3.7. Symbol $\mathcal{M}_{0.4}$ indicates MLP with a dropout rate of 0.4, $(Avg.)$ indicates the average performance of an optimizer over datasets, and $[Avg.]$ indicates the average performance of optimizers on a dataset. The classifier results of models MONT_3 and HFNT^{M} are from Ojha and Nicosia (2020). $\text{RF}^{\dagger}$ indicates random forest which is an ensemble model.

	Algorithm	Avg. classification accuracy $[1 - \mathcal{L}_{\text{Error}}(\cdot)]$										Avg. regression fit $\mathcal{L}_{r^2}(\cdot)$					
		Aus	Hrt	Ion	Pma	Wis	Irs	Win	Vhl	Gls	$(Avg.)$	Bas	Dee	Dia	Frd	Mpg	$(Avg.)$
BNeuralT ($\mathcal{G}$)	GD	.862	.848	.874	.789	.968	.947	.953	.671	.580	.832	.485	.748	.323	.653	.803	.602
	MGD	.886	.879	.935	.806	.980	.988	.980	.726	.687	.874	.585	.804	.434	.763	.849	.687
	NAG	.886	.878	.938	.808	.980	.987	.978	.731	.688	.875	.585	.804	.434	.757	.851	.686
	Adagrad	.872	.852	.907	.780	.974	.966	.981	.697	.638	.852	.621	.819	.432	.820	.851	.708
	RMSprop	**.895**	**.897**	**.952**	**.822**	**.986**	**.992**	**.991**	.750	**.732**	**.891**	.665	**.837**	**.492**	.776	**.867**	.727
	Adam	.875	.866	.870	.791	.982	.978	.974	.599	.657	.843	.579	.765	.360	.587	.825	.623
	$[Avg.]$	.880	.870	.913	.799	.978	.976	.976	.696	.663	.861	.587	.796	.412	.726	.841	.672
MLP ($\mathcal{M}$)	GD	.870	.831	.880	.763	.979	.968	.973	.806	.614	.854	.697	.821	.481	.736	.844	.716
	MGD	.874	.831	.907	.774	.981	.977	.985	**.853**	.629	.868	.718	.826	.486	.804	.852	.737
	NAG	.873	.828	.902	.772	.980	.976	.984	.852	.640	.868	.718	.826	.485	.786	.851	.733
	Adagrad	.870	.827	.723	.670	.944	.736	.877	.462	.362	.719	.407	.584	.221	.571	.630	.482
	RMSprop	.874	.832	.872	.758	.980	.969	.991	.804	.605	.854	.718	.826	.486	.866	.866	.752
	Adam	.876	.833	.882	.774	.984	.972	**.991**	.826	.635	.863	**.721**	.829	.490	**.943**	**.874**	**.772**
	$[Avg.]$	.873	.830	.861	.752	.975	.933	.967	.767	.581	.838	.663	.785	.442	.784	.820	.699
	$\mathcal{M}_{0.4}$Adam	.871	.815	.912	.774	.971	.960	.973	.806	.645	.859	.707	.828	.491	.884	.862	.754
Trees	HFNT^{S}	.824	.825	.871	.754	.973	.912	.918	.481	.544	.789	.576	.794	-0.062	.728	.775	.562
	HFNT^{M}	.826	.77	.822	.716	.935	.811	.824	.409	.399	.724	.608	.803	-0.13	.715	.838	.567
	MONT_3	.889	.809	.898	.799	.962	.989	.952	.55	.629	.831						
	DT	.801	.744	.890	.705	.932	.943	.897	.708	.698	.813	.431	.656	-0.12	.689	.756	.484
	$\text{RF}^{\dagger}$	.868	.809	.930	.752	.962	.959	.981	.745	**.768**	.864	.663	.829	.442	.871	.871	.735
	$[Avg.]$	.842	.791	.882	.745	.953	.923	.914	.579	.608	.804	.458	.605	.081	.601	.633	.476
Others	GP	.861	.820	.916	.769	.970	.960	.983	.843	.689	.868	.648	.820	.484	.724	.801	.695
	NBC	.797	.833	.882	.758	.930	.953	.969	.455	.457	.782						
	SVM	.861	.841	.880	.769	.977	.926	.978	.762	.575	.841	.647	.838	.405	.912	.861	.733
	$[Avg.]$	.840	.831	.893	.765	.959	.946	.977	.687	.574	.830	.648	.829	.445	.818	.831	.714
Parameter and time	$\mathbf{w}_{\mathcal{G}}$	204	560	185	157	268	180	358	249	190	261	140	163	140	141	178	152
	$\mathbf{w}_{\mathcal{M}} \vert \mathbf{w}_{\mathcal{M}_{0.4}}$	1702	1606	1602	3602	803	1102	2304	1703	3302	1969	1801	801	1201	701	701	1041
	$\tau^J_{\mathcal{G}}$	8.5	8.5	6.7	9.6	7.2	15.4	9.3	11.3	24.2	11.2	5.1	5.4	5.2	4.7	6.5	5.4
	$\tau^P_{\mathcal{M}}$	1173	802	628	860	410	1452	1206	412	4648	1288	1931	2031	1409	163	1074	1322
	τ^P_{DT}	1.0	4.2	1.9	1.1	1.5	6.4	5.6	1.2	4.5	3.1	2.0	2.6	1.8	0.6	2.0	1.8
	τ^P_{RF}	278.8	547.8	425.6	253.5	277.5	890.6	743.5	257.1	694.1	485.4	195.5	178.0	155.7	91.7	175.0	159.2
	τ^P_{GP}	187.2	58.6	108.7	195.8	197.6	226.1	253.7	2180	692.1	455.6	12.3	13.2	21.7	73.8	18.8	28.0
	τ^P_{NBC}	8.7	42.6	21.1	6.7	9.9	38.9	75.9	11.2	72.1	31.9						
	τ^P_{SVM}	5.8	20.4	15.5	4.5	5.0	36.7	25.9	5.3	25.6	16.1	17.6	15.5	30.7	9.2	12.2	17.1

Compared to that, a BNeuralT on MNIST (pixels) used 23,835 *trainable parameters* for an error rate of 6.08, and another model used 241,999 *trainable parameters* for an error rate of 5.19. Obviously, there is a trade-off between the model's parameters size and accuracy. The performances of a varied range of other algorithms on MNIST dataset are available at LeCun et al. (2020). Our goal is to use as much compact model as we can for high accuracy.

In our few trials, BNeuralT does perform competitively with many state-of-the-art (cf. Table 4). The performance of BNeuralT is better than tree-alternating optimization (TAO) (Carreira-Perpinan & Tavallali, 2018), CART (Breiman et al., 1984), C5.0 (Quinlan, 1993), oblique classifier 1 (OC1) (Murthy, Kasif, Salzberg, & Beigel, 1993), and generalized unbiased interaction detection and estimation (GUIDE) (Loh, 2014) algorithms that worked on MNIST raw pixels inputs (Zharmagambetov et al., 2019) like BNeuralT (cf. Table 4).

Table 3
Kolmogorov–Smirnov (KS) test on two samples: BNeuralT's RMSprop against all other algorithms for each data. The stat, pval, and post respectively indicate KS statistic, two-tailed p-value, and Bonferroni correction post-hoc adjusted p-value. The values are marked in bold where the *null hypothesis* that BNeuralT's RMSprop and other algorithms come from the same distribution is rejected as per Bonferroni correction.

BNeuralT's RMSprop vs.			Classification									Regression				
			Aus	Hrt	Ion	Pma	Wis	Irs	Win	Vhl	Gls	Bas	Dee	Dia	Frd	Mpg
MLP	GD	stat	.43	.63	.73	.70	.47	.67	.60	.63	.70	.27	.30	.23	.73	.43
		pval	.01	0	0	0	0	0	0	0	0	.24	.14	.39	0	.01
		post	.07	**0**	**0**	**0**	**.03**	**0**	**0**	**0**	**0**	1	1	1	**0**	.06
	MGD	stat	.40	.63	.53	.60	.43	.47	.30	.87	.60	.30	.20	.20	.20	.37
		pval	.02	0	0	0	.01	0	.14	0	0	.14	.59	.59	.59	.03
		post	.16	**0**	**0**	**0**	.07	.03	1	**0**	**0**	1	1	1	1	.31
	NAG	stat	.43	.63	.63	.57	.43	.47	.33	.87	.53	.30	.20	.17	.33	.33
		pval	.01	0	0	0	.01	0	.07	0	0	.14	.59	.81	.07	.07
		post	.07	**0**	**0**	**0**	.07	**.03**	.71	**0**	**0**	1	1	1	.64	.64
	Adagrad	stat	.43	.67	1	1	.87	1	.83	1	1	.83	1	.93	.93	1
		pval	.01	0	0	0	0	0	0	0	0	0	0	0	0	0
		post	.07	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	RMSprop	stat	.33	.60	.77	.67	.40	.50	.23	.60	.73	.30	.23	.20	.57	.20
		pval	.07	0	0	0	.02	0	.39	0	0	.14	.39	.59	0	.59
		post	.71	**0**	**0**	**0**	.16	**.01**	1	**0**	**0**	1	1	1	**0**	1
	Adam	stat	.30	.63	.67	.63	.47	.50	.17	.70	.63	.30	.17	.17	1	.20
		pval	.14	0	0	0	0	0	.81	0	0	.14	.81	.81	0	.59
		post	1	**0**	**0**	**0**	**.03**	**.01**	1	**0**	**0**	1	1	1	**0**	1
	$\mathcal{M}_{0.4}$Adam	stat	.37	.50	.53	.57	.40	.77	.30	.63	.70	.30	.40	.17	.50	.17
		pval	.03	0	0	0	.02	0	.14	0	0	.14	.02	.81	0	.81
		post	.45	**.01**	**0**	**0**	.20	**0**	1	**0**	**0**	1	.19	1	**.01**	1
Trees	HFNT^S	stat	.67	.70	.87	.70	.53	.53	.73	.93	.73	.47	.37		.40	.40
		pval	0	0	0	0	0	0	0	0	0	0	.03		.02	.02
		post	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	HFNT^M	stat	.67	.87	.83	.87	.63	.53	.77	1	.87	.43	.40		.63	.43
		pval	0	0	0	0	0	0	0	0	0	.01	.02		0	.01
		post	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	.85	.85
	DT	stat	.90	.87	.70	.93	.87	.67	.90	.53	.33	.67	.90	.93	.77	.87
		pval	0	0	0	0	0	0	0	0	.07	0	0	0	0	0
		post	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	.71	**0**	**0**	**0**	**0**	**0**
	RF	stat	.43	.73	.40	.90	.80	.57	.47	.30	.33	.30	.20	.37	.77	.20
		pval	.01	0	.02	0	0	0	0	.14	.07	.14	.59	.03	0	.59
		post	.09	**0**	.20	**0**	**0**	**0**	**.03**	1	.92	1	1	.41	**0**	1
Other	GP	stat	.43	.63	.57	.67	.43	.50	.33	.83	.30	.40	.37	.17	.73	.90
		pval	.01	0	0	0	.01	0	.07	0	.14	.02	.03	.81	0	0
		post	.07	**0**	**0**	**0**	.07	**.01**	.71	**0**	1	.14	.31	1	**0**	**0**
	NBC	stat	.93	.57	.77	.77	.93	.57	.50	1	.97					
		pval	0	0	0	0	0	0	0	0	0					
		post	**0**	**0**	**0**	**0**	**0**	**0**	**.01**	**0**	**0**					
	SVM	stat	.50	.53	.70	.67	.30	.70	.47	.20	.77	.37	.13	.50	.90	.20
		pval	0	0	0	0	.14	0	0	.59	0	.03	.96	0	0	.59
		post	**.01**	**0**	**0**	**0**	1	**0**	**.03**	1	**0**	.31	1	**.01**	**0**	1

We compare BNeuralT with biologically plausible models of Jones and Kording (2021) that performed binary classification on MNIST's two classes (class 3 and class 5). This is, however, a trivial comparison as BNeuralT works on all classes and uses sigmoidal dendritic nonlinearities, whereas Jones and Kording (2021)'s models work on binary class and use Leaky ReLU as dendritic nonlinearities. They obtained an error rate of 7.8%, 3.65%, and 8.89% respectively with 1-tree ($\mathbf{w} = 2047$), 32-tree ($\mathbf{w} = 65{,}504$), and A-32-tree ($\mathbf{w} = 65{,}504$) models. In contrast to Jones and Kording (2021)'s models, BNeuralT performs classification on all ten classes of MNIST pixels. Obviously, some classes are easier to learn than others (see Fig. 7), and training a binary classifier presents an entirely different difficulty level than a multi-class classification. However, although a one-to-one comparison is not possible in such a scenario, it may be worth noting that BNeuralT obtained an error rate of 7.74% ($\mathbf{w} = 11{,}987$) and 6.08% ($\mathbf{w} = 23{,}835$) on all ten classes. Therefore, the sparse stochastic structure of BNeuralT (e.g., Fig. 8) stands competitive with the models of Jones and Kording (2021).

Moreover, BNeuralT models show a linear relation between trainable parameters and their accuracy (cf. Table 4). Hence, BNeuralT models with relatively higher parameters and exhaustive hyperparameter tuning are able to produce efficient results. Fig. 7 shows an example BNeuralT (20K) model's training convergence and MNIST character classification performance on a receiver operating characteristic curve plot, where BNeuralT (20K) model for all classes produces a very *high sensitivity* (true-positive rate) and *very low specificity* (low false-positive rate). Of all classes, we may arrange classes on the scale of hardness of learnability in the order of "easiest to hardest" to learn as $c1$, $c6$, $c0$, $c7$, $c5$, $c4$, $c9$, $c2$, $c3$, and $c8$ (cf. Fig. 7).

5.3. BNeuralT convergence analysis

We evaluated average asymptotic convergence profiles of all six SGD optimizers for optimizing BNeuralT on classification and regression problems (cf. Figs. 9, 10, and 11). For such an analysis, we recorded training and test accuracies of each training epoch. Since we ran algorithms for 30 independent instances, we analyzed the average trajectory of all 30 runs. In each run, an ad hoc BNeuralT architecture was generated, which could vary in tree size between a minimum "outputs $\times$ 2" nodes to a maximum $(m^{p+1}-1)/(m-1)$ nodes. Hence, BNeuralT architecture and trainable parameters varied stochastically at each instance of the experiment. Such high entropy network architectures pose difficulties for SGDs to perform well consistently. We, therefore,

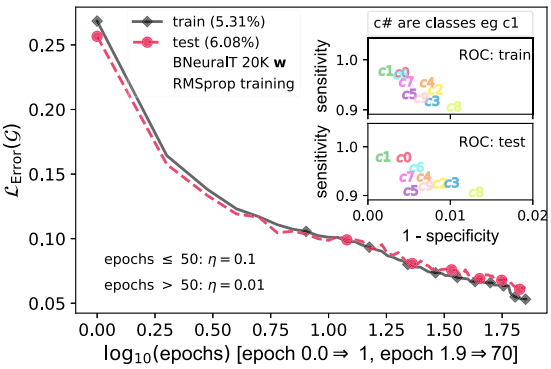


Fig. 7. BNeuralT-20K (23,835 trainable parameters **w**) model's RMSprop training and test error over 70 epochs on the MNIST dataset. Zoom-in for BNeuralT performance on receiver operating characteristic curve plots on training (*inner top*) and test (*inner bottom*) sets.

Table 4
Test error rate % of ad hoc BNeuralT ($\mathcal{G}$) models with a varied number of trainable parameters on the MNIST dataset. All models are trained for *70 epochs*, and where denoted † is for *25 epochs*. The decision tree models are reported in Zharmagambetov et al. (2019).

	Algorithms	Error(%)
BNeuralTs	BNeuralT-10K (pixels)	7.74
	BNeuralT-18K (pixels)	6.58
	BNeuralT-20K (pixels)	6.08
	BNeuralT-200K† (pixels)	5.19
Classification trees	GUIDE (pixels, oblique split)	26.21
	OC1 (pixels, oblique split)	25.66
	GUIDE (pixels)	21.48
	CART-R (pixels)	11.97
	CART-P (pixels)	11.95
	C5.0 (pixels)	11.69
	TAO (pixels)	11.48
	TAO (pixels, oblique split)	5.26

investigated BNeuralT models' accuracy against their architecture (number of parameters) (cf. Fig. 12).

BNeuralT classification models convergence. Fig. 9 shows convergence (training and test errors) profiles of ES training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate. With this BNeuralT's setting, we observe that BNeuralT's RMSprop converges the fastest among all SGDs. RMSprop also outperformed all other optimizers. NAG and MGD were asymptotically closer to RMSprop optimizer. Like RMSprop, NAG and MGD showed monotonically increasing training convergence. However, on the test sets, we observe that the models started overfitting. This motivated us to use early-stopping with restore best. Adagrad showed the most interesting convergence profile as initially, it had worse convergence among all optimizers, and while approaching higher epochs, it started rapidly improving its convergence. Thus, over an asymptotic behavior, Adagrad converged to a similar accuracy to that of RMSprop's accuracy. The optimizers NAG and MGD behave equivalently. Adam and GD were found sensitive to BNeuralT architecture (and trainable parameters).

BNeuralT regression models convergence. Fig. 10 shows convergence (training and test errors) profiles of ES training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate. The convergence profiles of optimizers on five regression problems suggest that RMSprop and Adagrad were better converging optimizers. Similar to its classification problems profile, RMSprop converged faster than other optimizers for regression problems. Adagrad showed slower convergence than RMSprop. However, unlike its performance on classification problems, Adagrad showed a more stable convergence profile for regression. Contrary to classification problems, overfitting occurred only occasionally for regression problems when comparing training and test convergence profiles.

BNeuralT and MLP settings convergence. In Fig. 11, we compare convergence of six optimizers for optimizing both BNeuralT and MLP on various settings. We show this comparison on "glass" and "miles per gallon" datasets as an example. (Supplementary shows convergence of all other datasets on various settings.) In Fig. 11, we observe that the learning rate 0.1 produces stable convergence for all optimizers. For learning rate 0.1, Adam does not converge as good as other algorithms (cf. Figs. 11(a) and 11(d)). However, Adam does converge when learning rate is 0.001 (cf. Figs. 11(b), 11(e), 11(h), and 11(k)). For learning rate 0.001, Adagrad does not converge as good as others. It may be observed that Adagrad's adaptively decreasing learning rate property made the convergence very slow and it may require more epochs to converge (cf. Figs. 11(b), 11(e), 11(h), and 11(k)).

Convergence of optimizers on ReLU. For *ReLU* activation function, Adagrad and GD being the slowest converging optimizers performed better than other faster converging optimizers like RMSprop, Adam, and NAG (cf. Figs. 11(c), 11(f), 11(l) and 11(l)). In fact, when using ReLU activation function for regression problems, BNeuralT suffered from exploding gradient issues when using optimizers like GD, MGD, NAG, and Adam during some instances of runs of some datasets. Adagrad, however, remained unaffected by exploding gradient issue. This is due to its decreasing convergence speed. BNeuralT's performance with ReLU, due to its high sparsity, was affected by exploding gradient effect more than the MLP, which showed more tolerance to exploding gradient effect due to its large number of parameters (cf. Supplementary Fig. A3).

Convergence of accuracy against trainable parameters. BNeuralT's tree size (proportional to trainable parameter) and test accuracy in Fig. 12 suggest that RMSprop compared to other optimizers can optimize ad hoc structure better. We observed that the accuracy of BNeuralT increases with increasing tree size. However, accuracy dropped for some outliers in the connected scatter plot in Fig. 12. This was because many points were within a specific range. For classification problems, except for RMSprop, NAG was another better optimizer. For regression problems, along with RMSprop, Adagrad was another better performing optimizer. BNeuralT's RMSprop optimizer showed rather more stable performance for stochastically varying architectures compared to other optimizers. For the pattern recognition MNIST dataset, RMSprop optimizer was used, and it showed a linear increase in accuracy for increasing order of tree size.

6. Discussion

We designed and investigated a learning system called BNeuralT capable of solving three classes of machine learning problems: classification, regression, and pattern recognition. We assessed the capability of this neural tree algorithm as a single neuron model approximating computational dendritic tree-like behavior (cf. Figs. 2 and 4). This algorithm can also be considered a highly sparse NN trained using SGD optimizers. To train BNeuralT using SGDs, we designed a recursive backpropagation algorithm. Therefore, we broadly assessed three aspects of a learning system, i.e., its performance on (i) stochastically generated highly sparse models, (ii) sigmoid and ReLU functions and their dendritic interactions with internal nodes, and (iii) optimizers asymptotic convergence behavior. We had a diverse

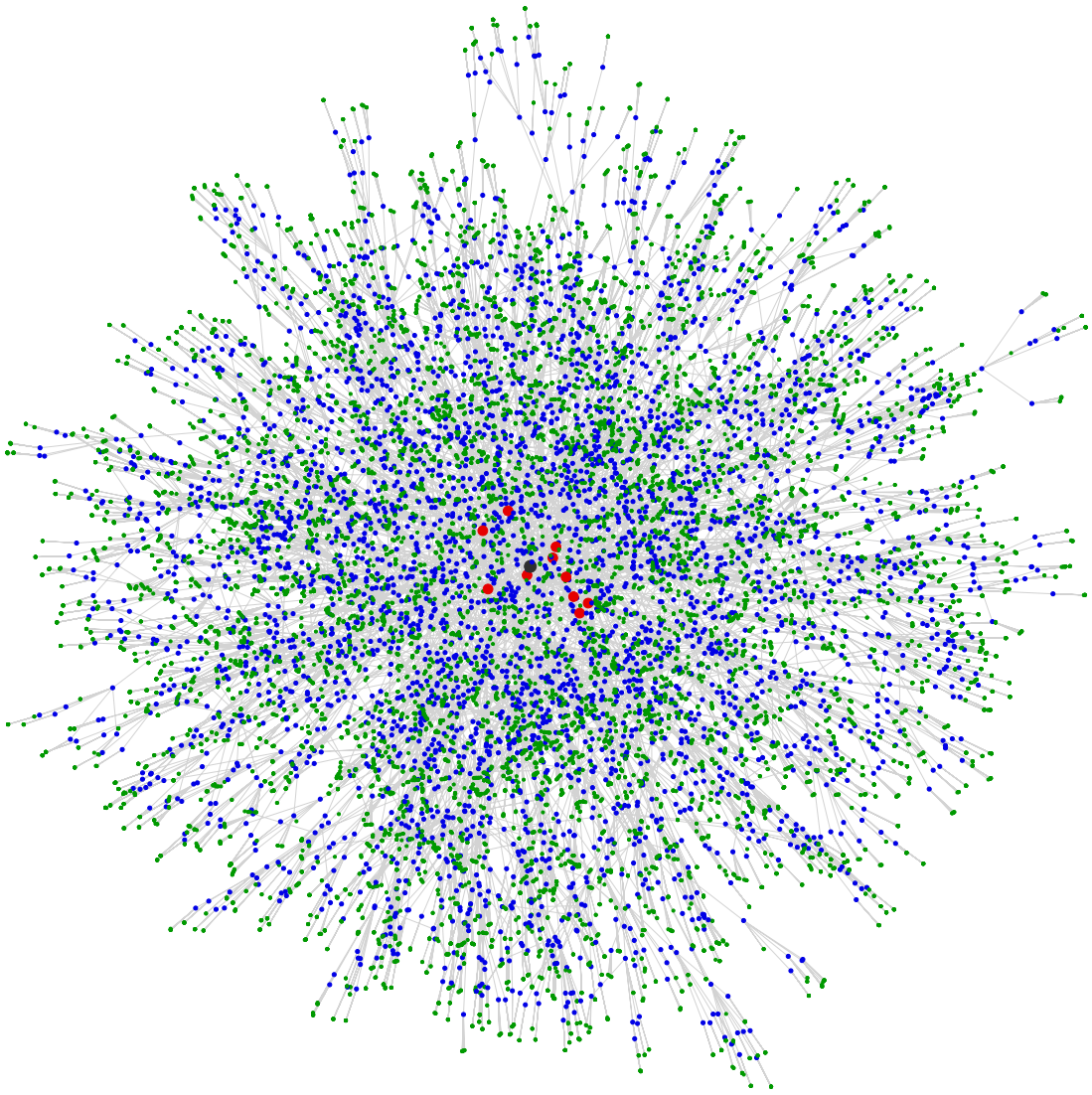


Fig. 8. BNeuralT-20K (pixels) MNIST model (tree structure). This model has 3664 function nodes (blue nodes), 16,507 leaf nodes (green nodes), and ten class nodes (red nodes in the inner circle), and the root node (in black) in the center. This model has 6738 edges (gray lines connecting nodes). These lines also represent neural weights. Each blue node also has its bias. Edge weights and bias together make 23,835 tree's trainable parameters. This model has a test accuracy of 94% (an error rate of 6.08%). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

range of classification and regression problems and algorithms to compare BNeuralT's capabilities over these dimensions.

Since BNeuralT resembles a highly sparse NN, its performance was assessed against a MLP (and a MLP with dropout rate similar to the probability of keeping nodes in BNeuralT) for their similar versions of SGD training. Six classification trees of BNeuralT, among all other algorithms and experiments, were top-performing models with a very low number of parameters. In fact, BNeuralT performed better against MLP dropout on classification problems, and it statistically had a similar performance on regression problems. This BNeuralT's performance against MLP's dropout regularization technique confirms that

> *stochastic gradient descent training of any a priori arbitrarily "thinned" network has the potential to solve machine learning tasks with equivalent or better degree of accuracy than a fully connected symmetric and systematic NN architecture.*

We used six different SGD optimizers for optimizing BNeuralT and MLP. Each optimizer behaved differently in terms of their asymptotic convergence depending on what problem they solve, how their learning rate behaved over the training epochs, and what activation function was used (cf. Figs. 9, 10, and 11). For example, with a 0.1 learning rate, RMSprop was the best among others for BNeuralT optimization over classification problems. For regression problems, both RMSprop and Adagrad performed well. Adagrad, however, was slow on classification problems. Since optimizers had to optimize the same architecture in an instance, it may be the continuous-variable output in the case of regression problems has helped Adagrad perform better than the discrete variable output in classification problems.

The use of activation functions influenced the performances of SGD optimizers. The sigmoid function proved to be more efficient with RMSprop, NAG, and MGD. Whereas ReLU proved to be efficient with Adagrad. This may be related to Adagrad's slow convergence speed that avoided weights to explode too quickly compared to faster converging optimizers like RMSprop (cf. Fig. 11(a–b), (d–e), (g–h)). This phenomenon of Adagrad may be confirmed since GD being the slowest converging SGD, was also found efficient when ReLU is used (cf. Fig. 11(c, f, and e)). Additionally, Adagrad converged better with a learning rate of 0.1 than 0.001 (e.g. Fig. 11(a–b)). This is because Adagrad was too slow at earlier epochs that prevented it from converging within a fixed number of training epochs.

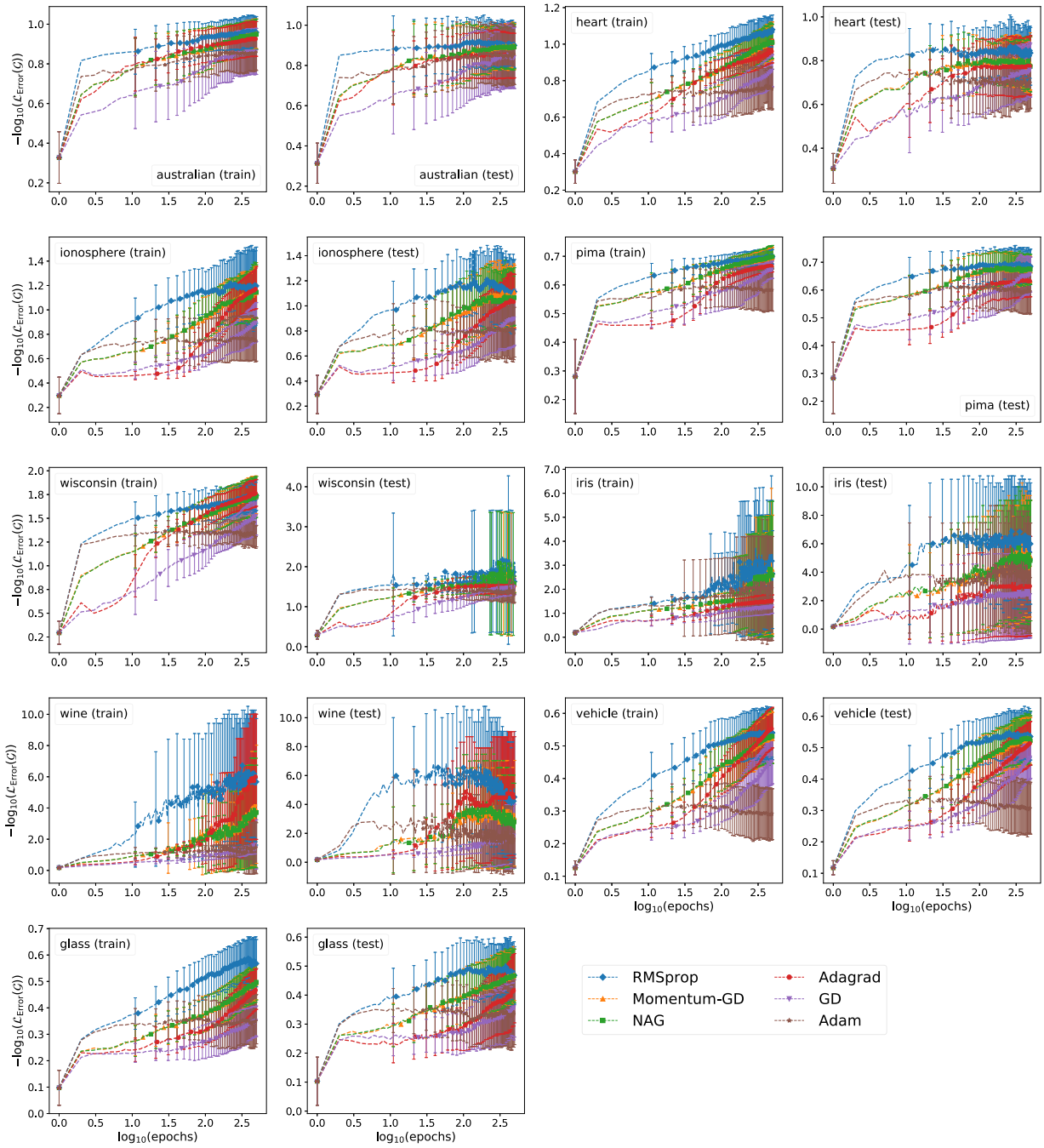


Fig. 9. Early-stopping training of BNeuralT having *sigmoid* nodes, 0.1 learning rate, and 0.4 leaf generation rate. BNeuralT average convergence trajectory performance computed over 30 independent runs for six optimizers over nine classification problems. The x-axis, $\log_{10}$(epochs) has the range $[0.0, 2.7]$ and is the training epochs 1 to 500. The y-axis, $-\log_{10}(\mathcal{L}_{\text{Error}}(\mathcal{G}))$ has the range $[0, 10]$ and is the log scale of the training and test accuracies. An error of 0.01 (accuracy 99%) on the $-\log_{10}(\mathcal{L}_{\text{Error}}(\mathcal{G}))$ scale has a value of 2.0 and an accuracy of 90% has a value of 1.0. Thus, a higher value on the y-axis is better. Error bar is the standard deviation of $-\log_{10}(\mathcal{L}_{\text{Error}}(\mathcal{G}))$ and it indicates stochasticity of the convergence that helps an optimizer escape local minima better. Thus, a larger length is better. For each data, training and test convergence pair are plotted for 500 epochs (on the log scale, 2.7). RMSprop, MGD, NAG, Adagrad, GD, and Adam are respectively indicated in blue, orange, green, red, purple, and brown colors, respectively, with symbols diamond, triangle, circle, downward triangle, and star. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

BNeuralT is operationally similar to HFNT and MONT algorithms. The HFNT and MONT algorithms model structures were genetically optimized as opposed to BNeuralT structure. The better performance of BNeuralT compared to HFNT and MONT shows that the stochastic structure of BNeuralT has a high potential to solve machine learning problems (cf. Table 2). However, this performance comparison also shows that BNeuralT models can be further compacted because both HFNT and MONT on classification problems had smaller average tree sizes than BNeuralT. This confirms that optimization of structure made HFNT and MONT more compact, although their accuracies were slightly compromised. On regression problems, however, BNeuralT performed better than HFNT both in terms of tree size and regression fit.

BNeuralT's performance compared to MLP's (with and without dropout) models and genetically optimized HFNT and MONT models confirms *Occam's razor principle of parsimony* for machine learning model selection that the simple models possess better generalization capability than the complex models (Blumer,

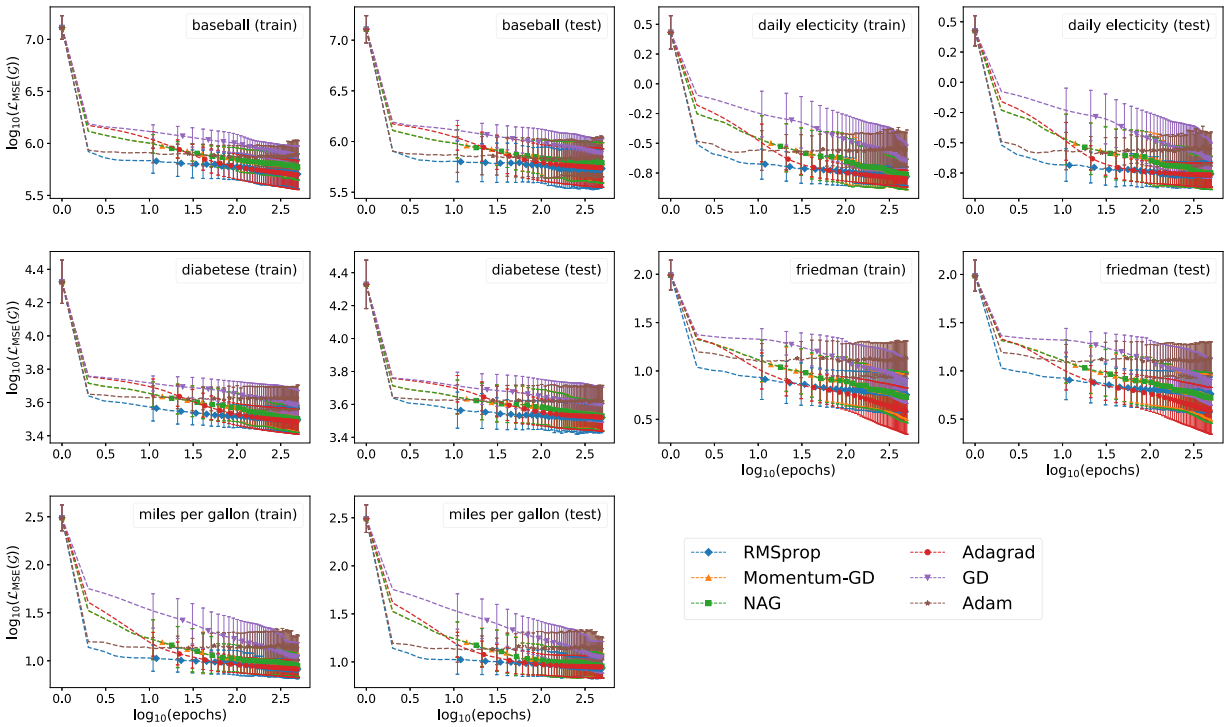


Fig. 10. BNeuralT average convergence trajectory performance computed over 30 independent runs for six optimizers over five regression problems. The x-axis, $\log_{10}(\text{epochs})$ has the range $[0.0, 2.7]$ and is the training epochs 1 to 500. The y-axis, $\log_{10}(\mathcal{L}_{\text{MSE}}(\mathcal{G}))$ is the training and test sets mean square error (MSE) on the log scale. An MSE 0.01 on the log scale has a value of -2.0. Thus, a lower value is better. Error bar is the standard deviation of $\log_{10}(\mathcal{L}_{\text{MSE}}(\mathcal{G}))$ and it indicates stochasticity of the convergence that helps an optimizer escape local minima better. Thus, a larger length is better. For each dataset, training and test convergence pair are plotted for 500 epochs (on the log scale, 2.7). The optimizers RMSprop, MGD, NAG, Adagrad, GD, and Adam are respectively indicated in blue, orange, green, red, purple, and brown colors with respective symbols diamond, triangle, circle, down triangle, and star. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Ehrenfeucht, Haussler, & Warmuth, 1987). Indeed, it is similar to the *sparsity of the biological brain* that a sparse network generalizes better or as good as a dense network (Friston, 2008; Herculano-Houzel, Mota, Wong, & Kaas, 2010; Hoefler, Alistarh, Ben-Nun, Dryden, & Peste, 2021). Moreover, it has been argued that a dense network is often overparameterized, and only a minute fraction of it is required for generalization (Denil, Shakibi, Dinh, Ranzato, & De Freitas, 2013). Our result is in a similar line because BNeuralT, with only an average of 222 parameters, which is *only* 13.5% of parameters than that of MLP's average 1638 parameters, is able to generalize machine learning problems better or with similar accuracy than MLP. Additionally, the sparsity and compactness of BNeuralT models reduce memory usage and CO_2 footprint as they require less memory and computational resources than dense networks.

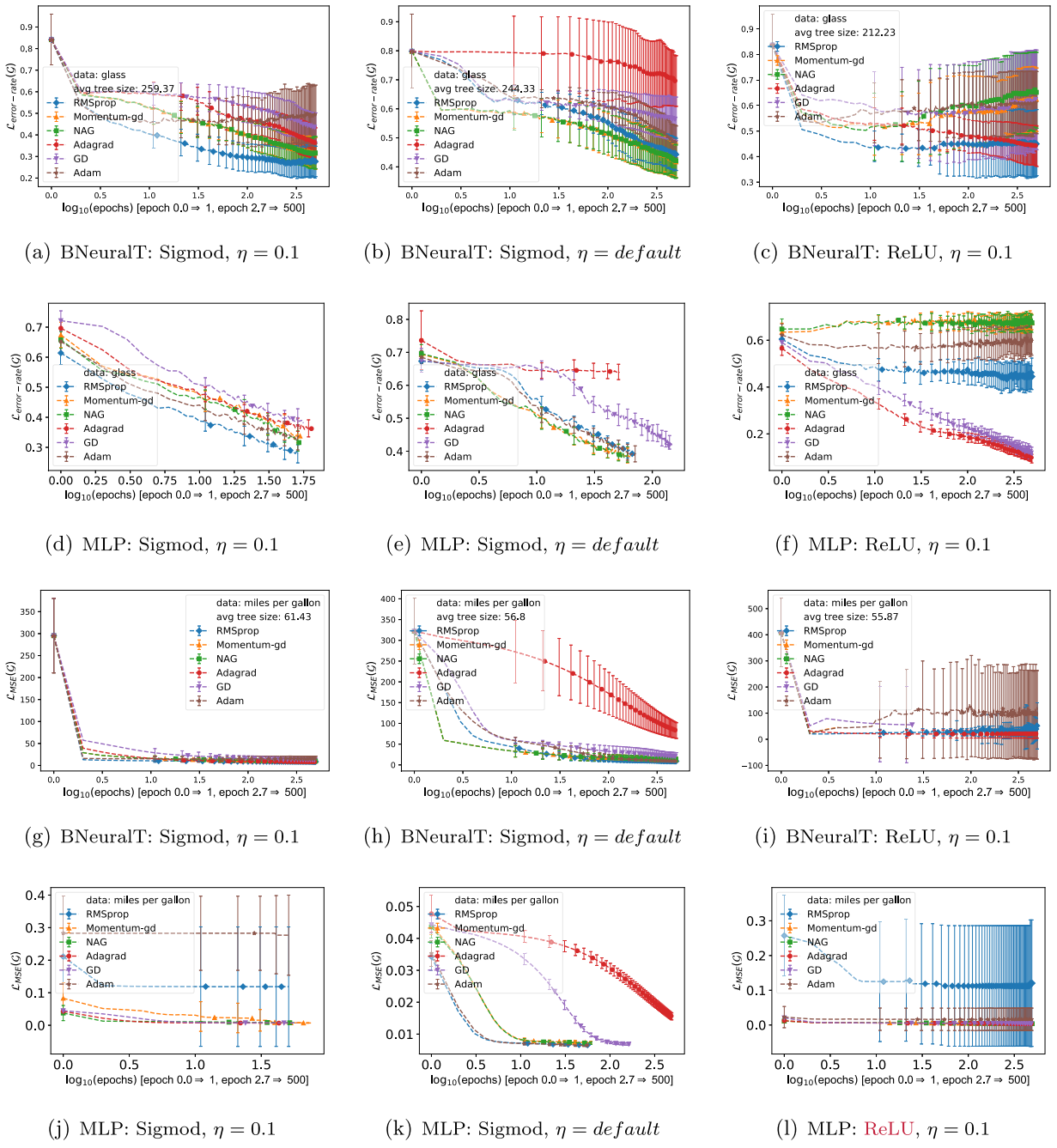
The decision tree algorithms DT and RF (ensemble of DTs) computationally have dedicated paths from the root to leaves (Breiman, 2001; Breiman et al., 1984). BNeuralT computationally also has dedicated information processing paths but from leaves to root. Although these algorithms differ in how nodes propagate information, a performance comparison suggests that BNeuralT has superior or competitive performances compared with DT and RF (cf. Table 2). This performance is noticeable since RF is an ensemble algorithm that, using bootstrapping, combines 100 DTs to construct a predictor (Breiman, 2001). Hence, the better performance of a standalone randomly generated BNeuralT model shows its high capabilities. Especially when RF being an ensemble of many trees, is more complex than a small and compact BNeuralT tree. Moreover, DTs are symbolic machine learning algorithms whose models offer inference ability as opposed to the black-box nature of NNs because of their ability to induce data using dedicated paths from the root to leaves. Likewise, as shown in Figs. 5 and 6,

BNeuralT has dedicated information processing paths from leaves to root, and such paths related to particular subsets of inputs may be analyzed, which potentially may offer inference ability to BNeuralT.

Thus, BNeuralT models are potentially inferable as opposed to NNs. However, this is a challenging task since BNeuralT's nodes combine inputs and perform a nonlinear or linear transformation.

We assessed BNeuralT performance against GP, NBC, and SVM. These three algorithms take Gaussian kernels. That is, these algorithms have a powerful approach towards prediction. GP and NBC algorithms are robust and powerful algorithms if input data follow a normal distribution. Similarly, SVM uses Gaussian kernels to project input to high dimensions, increasing the separability of data points to help to classify them (Cortes & Vapnik, 1995). Better performance of BNeuralT compared to these algorithms on classification and regression problems (cf. Table 2) suggests that BNeuralT offers an efficient alternative to these algorithms as BNeuralT does not make any assumption about data to generate a hypothesis (model) when fitting or classifying data.

The biologically plausible design of BNeuralT comes from its structural arrangement that takes random repeated input and has a computational dendritic tree-like organization with sigmoidal nonlinearities or ReLU linearity through its internal nodes (London & Häusser, 2005). The biologically plausible computational dendritic tree-like models 1-tree and 32-tree have a regular structural arrangement where repeated inputs are fed to a neuron systematically to form a tree structure (Jones & Kording, 2021). Whereas BNeuralT takes randomly generated inputs and takes a non-systematic stochastic approach to its tree construction (cf. Fig. 2). Moreover, BNeuralT works on multi-class classification; and 1-tree, 32-tree, and A-32-tree models work on binary classification (Jones & Kording, 2021).



(a) BNeuralT: Sigmod, $\eta = 0.1$ (b) BNeuralT: Sigmod, $\eta = default$ (c) BNeuralT: ReLU, $\eta = 0.1$

(d) MLP: Sigmod, $\eta = 0.1$ (e) MLP: Sigmod, $\eta = default$ (f) MLP: ReLU, $\eta = 0.1$

(g) BNeuralT: Sigmod, $\eta = 0.1$ (h) BNeuralT: Sigmod, $\eta = default$ (i) BNeuralT: ReLU, $\eta = 0.1$

(j) MLP: Sigmod, $\eta = 0.1$ (k) MLP: Sigmod, $\eta = default$ (l) MLP: ReLU, $\eta = 0.1$

Fig. 11. Comparison of convergence of optimizers for optimizing BNeuralT and MLP and optimizers convergence over varied learning rate settings and activation function usage. (a)–(f) Classification problem where y-axis is an error rate. (g)–(l) Regression problems where y-axis is an MSE. The x-axis, $\log_{10}$(epochs) has the range $[0.0, 2.7]$ and is the training epochs 1 to 500. For MLP, early-stopping method show values for optimizes only upto the epochs where training stopped. Learning rate η value *default* indicates that RMSprop, Adam, and Adagrad has a value of 0.001 as their learning rate and GD, NAG, and MGD has a value of 0.01.

BNeuralT's comparison with 1-tree, 32-tree, A-32-tree models, although limited, presents a noticeable performance. The error rate of BNeuralT on the MNIST dataset on all ten classes classification was 6.08% with 23 835 parameters. The error rates of 1-tree, 32-tree, A-32-tree models on the binary classification of classes 3 and 5 of the MNIST dataset were reported as 7.8%, 3.65%, and 8.89%, respectively, and they had 2047, 65 504, 65 504 parameters, respectively. This result confirms BNeuralT's potential to produce capable learning systems, especially when BNeuralT's structural randomness (cf. Fig. 2) is closer to the randomness (if any) of biological computational dendritic-tree (Travis et al., 2005).

7. Conclusions

We propose a new algorithm Backpropagation Neural Tree (BNeuralT). Our BNeuralT algorithm plausibly has a *biological dendritic tree-like* modeling capability. It has a *single neuron-like* model with sigmoidal dendritic nonlinearities or rectified linear unit (ReLU) based dendritic linearity. It uses random repeated inputs at the leaves of subtrees attached to a single neuron, which is the root of a tree. BNeuralT uses stochastic gradient descent (SGD) optimizers to optimize stochastically generated sparse tree structures that are potentially minimal subsets of neuron networks (NNs). We propose a *recursive error backpropagation*

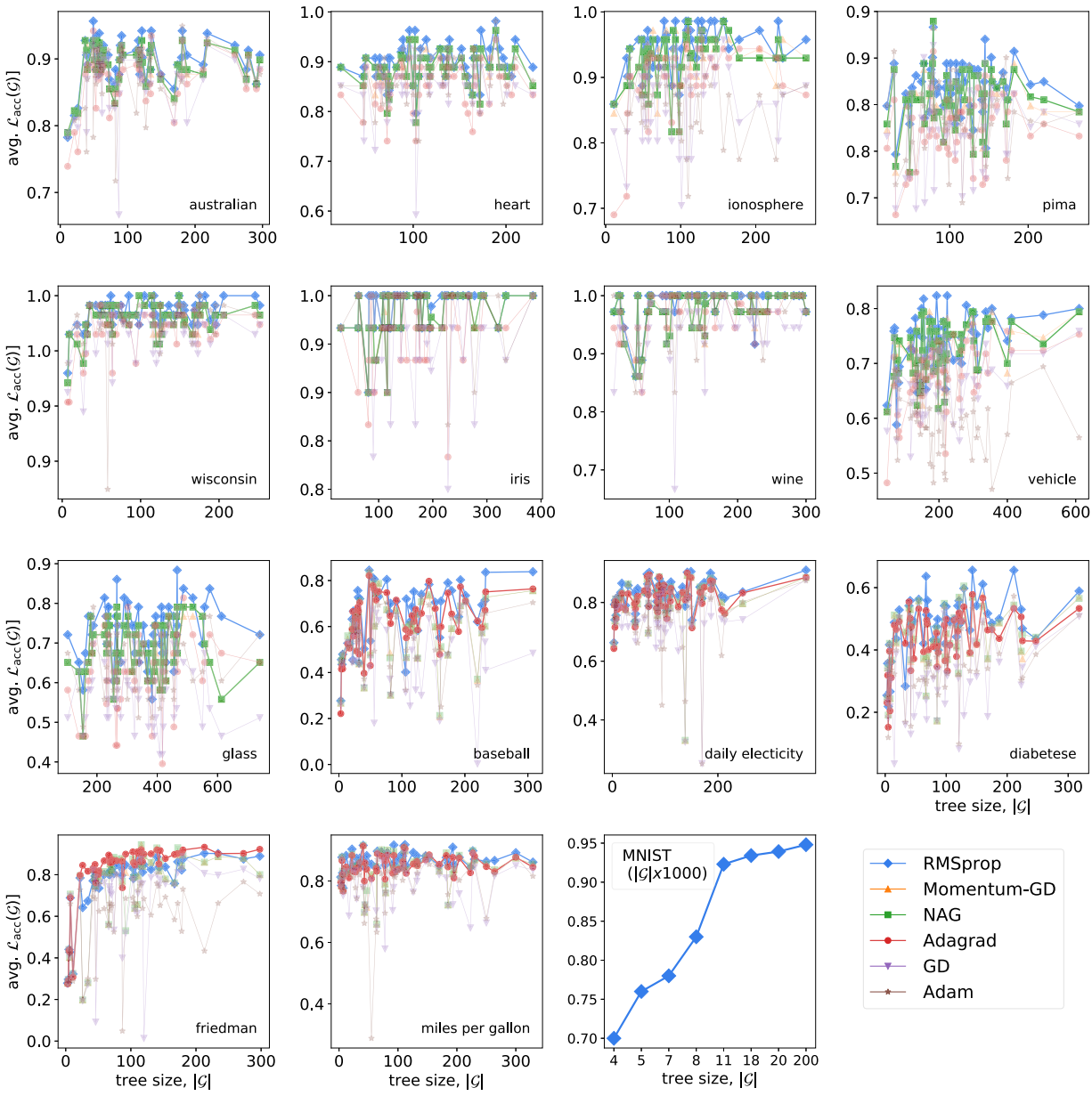


Fig. 12. BNeuralT model's tree size $|\mathcal{G}|$ (x-axis) compared with its average (avg.) accuracy $\mathcal{L}_{acc}(\mathcal{G}) = (1 - \mathcal{L}_{Error}(\mathcal{G}))$ (y-axis has range $[0, 1]$) on the training sets of nine classification problems, five regression problems, and one pattern recognition problem. For the pattern recognition problem, BNeuralT model's tree size $|\mathcal{G}|$ (x-axis) is in the multiple of 1000. The optimizers RMSprop, MGD, NAG, Adagrad, GD, and Adam are respectively indicated in blue, orange, green, red, purple, and brown colors, respectively, with symbols diamond, triangle, circle, downward triangle, and star. For a few cases, convergence is linear to tree size, however for a few, high accuracy is achieved with smaller trees. For MNIST, RMSprop has 10-epoch of stochastic online training and shows a linear relation with tree size. In all tasks, the convergence of RMSprop (blue diamonds) is the best, followed by NAG (green squares) and MGD (yellow triangles). Adagrad (red circles) shows competitive performance with RMSprop (blue diamonds) for regression problems. Classification datasets have green and blueish hues because NAG and RMSprop are top optimizers, and for regression datasets, plots appear to be red and blueish hue because of RMSprop and Adagrad are top optimizers. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

algorithm to apply SGDs to train trees that require pre-order and post-order traversal in a depth-first-search manner for their forward pass and backward pass computations.

The results show that our stochastically generated biologically plausibly tree structure and recursive error backpropagation algorithm have the capacity to learn a wide variety of machine learning problems. Moreover, we show that any stochastically generated tree structures can learn machine learning problems with high accuracy, and structure optimization may only be required for making models more compact. However, there is a trade-off for compacting models as we found that making the models more compact means compromising on accuracy. Additionally, BNeuralT's strong performance compared to MLP's dropout regularization technique confirms that SGD training of any "*a priori*" arbitrarily "*thinned network*" (spares tree structures) has the potential to solve machine learning tasks with equivalent or better degree of accuracy.

The sigmoidal dendritic nonlinearities (sigmoid function used at tree's root and internal nodes) performed obviously better than a linear dendritic tree (sigmoid function used at tree's root and ReLU at internal nodes). However, the linear dendritic tree differed from the best performing nonlinear dendritic tree by only about 10% accuracy. Nevertheless, it was comparable with a few nonlinear dendritic tree models, especially with those trained with gradient descent (GD), momentum GD, and Adam. This

shows that *purely single node* BNeuralT models might solve machine learning problems efficiently.

On MNIST (pixels) character classification dataset, BNeuralT, when loosely compared with 1-tree and 32-tree biologically plausible dendritic tree algorithms, was found competitive. Moreover, BNeuralT performed best among select tree-based classifiers for the classification of MNIST characters. On classification and regression problems, the overall performance of BNeuralT was better than some varied types of well-known algorithms: decision tree, random forest, Gaussian process, naïve Bayes classifier, and support vector machine. Such a performance of BNeuralT came from a minimal hyperparameter setup. Therefore, this work shows that our newly designed learning algorithm generates high-performing and parsimonious (therefore sustainable) models balancing the complexity with descriptive ability.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.neunet.2022.02.003. Supplementary provides additional plots, statistical tests, hyperparameters table, data, and source code links.

References

Bache, K., & Lichman, M. (2013). UCI machine learning repository. https://archive.ics.uci.edu/ml/index.php (Accessed on: 01 Apr 2020).

Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2013). Advances in optimizing recurrent networks. In *IEEE international conference on acoustics, speech and signal processing* (pp. 8624–8628). IEEE.

Beniaguev, D., Segev, I., & London, M. (2020). Single cortical neurons as deep artificial neural networks. *BioRxiv*, Article 613141.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters*, *24*(6), 377–380.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). *Classification and regression trees*. CRC Press.

Carreira-Perpinan, M. A., & Tavallali, P. (2018). Alternating optimization of decision trees, with application to learning sparse oblique trees. In *Advances in neural information processing systems, Vol. 31* (pp. 1211–1221). Curran Associates, Inc..

Chang, C.-C., & Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *2*(3), 1–27.

Chen, Y., Yang, B., & Abraham, A. (2007). Flexible neural trees ensemble for stock index modeling. *Neurocomputing*, *70*(4–6), 697–703.

Chen, Y., Yang, B., Dong, J., & Abraham, A. (2005). Time-series forecasting using flexible neural tree model. *Information Sciences*, *174*(3), 219–235.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273–297.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems* (pp. 1223–1231).

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., & De Freitas, N. (2013). Predicting parameters in deep learning. In *Advances in neural information processing systems*.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., & Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, *9*, 1871–1874.

Farhoodi, R., & Kording, K. P. (2018). Sampling neuron morphologies. *BioRxiv*, Article 248385.

Friston, K. (2008). Hierarchical models in the brain. *PLoS Computational Biology*, *4*(11), Article e1000211.

Hay, E., Hill, S., Schürmann, F., Markram, H., & Segev, I. (2011). Models of neocortical layer 5b pyramidal cells capturing a wide range of dendritic and perisomatic active properties. *PLoS Computational Biology*, *7*(7), Article e1002107.

Herculano-Houzel, S., Mota, B., Wong, P., & Kaas, J. H. (2010). Connectivity-driven white matter scaling and folding in primate cerebral cortex. *Proceedings of the National Academy of Sciences*, *107*(44), 19008–19013.

Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, *117*(4), 500–544.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., & Peste, A. (2021). Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. arXiv:2102.00554.

Jones, I. S., & Kording, K. P. (2021). Might a single neuron solve interesting machine learning problems through successive computations on its dendritic tree? *Neural Computation*, *33*(6), 1554–1571.

Jordan, M. I., & Jacobs, R. A. (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, *6*(2), 181–214.

Keel (2011). KEEL dataset repository. https://sci2s.ugr.es/keel/datasets.php (Accessed on: 01 Apr 2020).

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *Proc. of ICNN'95-International conference on neural networks, Vol. 4* (pp. 1942–1948). IEEE.

Keras (2020). Keras the sequential model. https://keras.io/guides/sequential_model/ (Accessed on: 01 Apr 2020).

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd international conference for learning representations (ICLR)*.

LeCun, Y., Cortes, C., & Burges, C. J. (2020). The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/ (Accessed on: 01 Apr 2020).

Lee, C.-Y., Gallagher, P. W., & Tu, Z. (2016). Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In *Artificial intelligence and statistics* (pp. 464–472).

Loh, W.-Y. (2014). Fifty years of classification and regression trees. *International Statistical Review*, *82*(3), 329–348.

London, M., & Häusser, M. (2005). Dendritic computation. *Annual Review of Neuroscience*, *28*, 503–532.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

Mel, B. W. (2016). Toward a simplified model of an active dendritic tree. In G. Stuart, N. Spruston, & M. Häusser (Eds.), *Dendrites* (pp. 405–486). chapter 16.

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill.

Murthy, S., Kasif, S., Salzberg, S., & Beigel, R. (1993). OC1: A randomized induction of oblique decision trees. In *Proc. 11th national conference on artificial intelligence (AAAI), Vol. 93* (pp. 322–327).

Ojha, V. K., Abraham, A., & Snášel, V. (2017). Ensemble of heterogeneous flexible neural trees using multiobjective genetic programming. *Applied Soft Computing*, *52*, 909–924.

Ojha, V., & Nicosia, G. (2020). Multi-objective optimisation of multi-output neural trees. In *IEEE congress on evolutionary computation (CEC)* (pp. 1–8). IEEE.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., et al. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering* (pp. 256–267).

Poirazi, P., Brannon, T., & Mel, B. W. (2003a). Arithmetic of subthreshold synaptic summation in a model CA1 pyramidal cell. *Neuron*, *37*(6), 977–987.

Poirazi, P., Brannon, T., & Mel, B. W. (2003b). Pyramidal neuron as two-layer neural network. *Neuron*, *37*(6), 989–999.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, *12*(1), 145–151.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. Morgan Kaufmann.

Rasmussen, C. E., & Williams, C. (2006). *Gaussian processes for machine learning*. MIT Press.

Rios, L. M., & Sahinidis, N. V. (2013). Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, *56*(3), 1247–1293.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*.

Sabour, S., Frosst, N., & Hinton, G. E. (2017). Dynamic routing between capsules. In *Advances in neural information processing systems* (pp. 3856–3866).

Sakar, A., & Mammone, R. J. (1993). Growing and pruning neural tree networks. *IEEE Transactions on Computers*, *42*(3), 291–299.

Schmidt, M. D., & Lipson, H. (2009). Solving iterated functions using genetic programming. In *Proc. 11th annual conference companion on genetic and evolutionary computation conference: Late breaking papers* (pp. 2149–2154). ACM.

Sirat, J., & Nadal, J. (1990). Neural trees: a new tool for classification. *Network. Computation in Neural Systems*, *1*(4), 423–438.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(56), 1929–1958.

Tanno, R., Arulkumaran, K., Alexander, D., Criminisi, A., & Nori, A. (2019). Adaptive neural trees. In *Proc. 36th international conference on machine learning (ICML)* (pp. 6166–6175).

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *Coursera: Neural Networks for Machine Learning*, *4*(2), 26–31.

Travis, K., Ford, K., & Jacobs, B. (2005). Regional dendritic variation in neonatal human cortex: a quantitative Golgi study. *Developmental Neuroscience*, *27*(5), 277–287.

Wolpert, D. H. (1996). The lack of a priori distinctions between learning algorithms. *Neural Computation*, *8*(7), 1341–1390.

Zhang, B.-T., Ohm, P., & Mühlenbein, H. (1997). Evolutionary induction of sparse neural trees. *Evolutionary Computation*, *5*(2), 213–236.

Zharmagambetov, A., Hada, S. S., Carreira-Perpiñán, M. A., & Gabidolla, M. (2019). An experimental comparison of old and new decision tree algorithms. arXiv: 1911.03054.