The Tree Ensemble Layer: Differentiability meets Conditional Computation

Hussein Hazimeh [1] Natalia Ponomareva [2] Petros Mol [2] Zhenyu Tan [3] Rahul Mazumder [1]

Abstract

Neural networks and tree ensembles are state-of-the-art learners, each with its unique statistical and computational advantages. We aim to combine these advantages by introducing a new layer for neural networks, composed of an ensemble of differentiable decision trees (a.k.a. soft trees). While differentiable trees demonstrate promising results in the literature, in practice they are typically slow in training and inference as they do not support conditional computation. We mitigate this issue by introducing a new sparse activation function for sample routing, and implement true conditional computation by developing specialized forward and backward propagation algorithms that exploit sparsity. Our efficient algorithms pave the way for jointly training over deep and wide tree ensembles using first-order methods (e.g., SGD). Experiments on 23 classification datasets indicate over 10x speed-ups compared to the differentiable trees used in the literature and over 20x reduction in the number of parameters compared to gradient boosted trees, while maintaining competitive performance. Moreover, experiments on CIFAR, MNIST, and Fashion MNIST indicate that replacing dense layers in CNNs with our tree layer reduces the test loss by 7-53% and the number of parameters by 8x. We provide an open-source TensorFlow implementation with a Keras API.

1. Introduction

Decision tree ensembles have proven very successful in various machine learning applications. Indeed, they are often referred to as the best "off-the-shelf" learners (Hastie et al., 2009), as they exhibit several appealing properties such as ease of tuning, robustness to outliers, and interpretability (Hastie et al., 2009; Chen & Guestrin, 2016). Another natural property in trees is *conditional computation*, which refers to their ability to route each sample through a small number of nodes (specifically, a single root-to-leaf path). Conditional computation can be broadly defined as the ability of a model to activate only a small part of its architecture in an input-dependent fashion (Bengio et al., 2015). This leads to both computational benefits and enhanced statistical properties. On the computation front, routing samples through a small part of the tree leads to substantial training and inference speed-ups compared to methods that do not route samples. Statistically, conditional computation offers the flexibility to reduce the number of parameters used by each sample, thus acting as a regularizer (Breiman et al., 1983; Hastie et al., 2009; Bengio et al., 2015).

However, the performance of trees relies on feature engineering, as they lack a good mechanism for representation learning (Bengio et al., 2013). This is an area in which neural networks (NNs) excel, especially in speech and image recognition applications (Bengio et al., 2013; He et al., 2015; Yu & Deng, 2016). However, NNs do not naturally support conditional computation and are harder to tune.

In this work, we combine the advantages of neural networks and tree ensembles by designing a hybrid model. Specifically, we propose the *Tree Ensemble Layer (TEL)* for neural networks. This layer is an additive model of differentiable decision trees, can be inserted anywhere in a neural network, and is trained along with the rest of the network using gradient-based optimization methods (e.g., SGD). While differentiable trees in the literature show promising results, especially in the context of neural networks, e.g., (Kontschieder et al., 2015; Frosst & Hinton, 2017), they do not offer true conditional computation. We equip TEL with a novel mechanism to perform conditional computation, during both training and inference. We make this possible by introducing a new sparse activation function for sample routing, along with specialized forward and backward propagation algorithms that exploit sparsity. Experiments on 23 real datasets indicate that TEL achieves over 10x speed-ups compared to the current differentiable trees, without sacrificing predictive performance.

Our algorithms pave the way for jointly optimizing over both wide and deep tree ensembles. Here joint optimization refers to updating all the trees simultaneously (e.g., using first-order methods like SGD). This has been a major computational challenge prior to our work. For example, jointly

[1] Massachusetts Institute of Technology [2] Google Research [3] Google Brain. Correspondence to: Hussein Hazimeh <hazimeh@mit.edu>.

optimizing over classical (non-differentiable) decision trees is a hard combinatorial problem (Hastie et al., 2009). Even with differentiable trees, the training complexity grows exponentially with the tree depth, making joint optimization difficult (Kontschieder et al., 2015). A common approach is to train tree ensembles using greedy "stage-wise" procedures, where only one tree is updated at a time and never updated again—this is a main principle in gradient boosted decision trees (GBDT) (Friedman, 2001). We hypothesize that joint optimization yields more compact and expressive ensembles than GBDT. Our experiments confirm this, indicating that TEL can achieve over 20x reduction in model size. This can have important implications for interpretability, latency and storage requirements during inference.

Contributions: Our contributions can be summarized as follows: **(i)** We design a new differentiable activation function for trees which allows for routing samples through small parts of the tree (similar to classical trees). **(ii)** We realize conditional computation by developing specialized forward and backward propagation algorithms that exploit sparsity to achieve an optimal time complexity. Notably, the complexity of our backward pass can be independent of the tree depth and is generally better than that of the forward pass—this is not possible in backpropagation for neural networks. **(iii)** We perform experiments on a collection of 26 real datasets, which confirm TEL as a competitive alternative to current differentiable trees, GBDT, and dense layers in CNNs. **(iv)** We provide an open-source TensorFlow implementation of TEL along with a Keras interface[1].

Related Work: Table 1 summarizes the most relevant related work. Differentiable decision trees (a.k.a. soft trees) were introduced by (Jordan & Jacobs, 1994). The internal nodes of these trees act as routers, sending samples to the left and right with different proportions. This framework does not support conditional computation as each sample is processed in all the tree nodes. Our work avoids this issue by allowing each sample to be routed through small parts of the tree, without losing differentiability. A number of recent works have used soft trees in the context of deep learning. For example, (Kontschieder et al., 2015) equipped soft trees with neural representations and used alternating minimization to learn the feature representations and the leaf outputs. (Hehn et al., 2019) extended (Kontschieder et al., 2015)'s approach to allow for conditional inference and growing trees level-by-level. (Frosst & Hinton, 2017) trained a (single) soft tree using SGD and leveraged a deep neural network to expand the dataset used in training the tree. (Zoran et al., 2017) also leveraged a tree structure with a routing mechanism similar to soft trees, in order to equip the k-nearest neighbors algorithm with neural representations. All of these works have observed that computation in a soft tree can be expensive. Thus, in practice, heuristics are used to speed up inference, e.g., (Frosst & Hinton, 2017) uses the root-to-leaf path with the highest probability during inference, leading to discrepancy between the models used in training and inference. Instead of making a tree differentiable, (Jernite et al., 2017) hypothesized about properties the best tree should have, and introduced a pseudo-objective that encourages balanced and pure splits. They optimized using SGD along with intermediate processing steps.

Table 1: Related work for conditional computing

Paper	CT	CI	DO	Model/Optim
(Kontschieder et al., 2015)	N	N	Y	Soft tree/Alter
(Ioannou et al., 2016)	N	H	Y	Tree-NN/SGD
(Frosst & Hinton, 2017)	N	H	Y	Soft tree/SGD
(Zoran et al., 2017)	N	H	N	Soft tree/Alter
(Shazeer et al., 2017)	H	Y	N	Tree-NN/SGD
(Tanno et al., 2018)	N	H	Y	Soft tree/SGD
(Biau et al., 2019)	H	N	Y	Tree-NN/SGD
(Hehn et al., 2019)	N	H	Y	Soft tree/SGD
Our method	Y	Y	Y	Soft tree/SGD

H is heuristic (e.g., training model is different from inference), *CT* is conditional training. *CI* is conditional inference. *DO* indicates whether the objective function is differentiable. *Soft tree* refers to a differentiable tree, whereas *Tree-NN* refers to NNs with a tree-like structure. *Optim* stands for optimization (SGD or alternating minimization).

Another line of work introduces tree-like structure to NNs via some routing mechanism. For example, (Ioannou et al., 2016) employed tree-shaped CNNs with branches as weight matrices with sparse block diagonal structure. (Shazeer et al., 2017) created the Sparsely-Gated Mixture-of-Experts layer where samples are routed to subnetworks selected by a trainable gating network. (Biau et al., 2019) represented a decision tree using a 3-layer neural network and combined CART and SGD for training. (Tanno et al., 2018) looked into adaptively growing an NN with routing nodes for performing tree-like conditional computations. However, in these works, the inference model is either different from training or the router is not differentiable (but still trained using SGD)—see Table 1 for details.

2. The Tree Ensemble Layer (TEL)

TEL is an additive model of differentiable decision trees. In this section, we introduce TEL formally and then discuss the routing mechanism used in our trees. For simplicity, we assume that TEL is used as a standalone layer. Training trees with other layers will be discussed in Section 3.

We assume a supervised learning setting, with input space $\mathcal{X} \subseteq \mathbb{R}^p$ and output space $\mathcal{Y} \subseteq \mathbb{R}^k$. For example, in the case of regression (with a single output) $k = 1$, while in

[1] https://github.com/google-research/google-research/tree/master/tf_trees

classification k depends on the number of classes. Let m be the number of trees in the ensemble, and let $T^{(j)} : \mathcal{X} \to \mathbb{R}^k$ be the jth tree in the ensemble. For an input sample $x \in \mathbb{R}^p$, the final output of the layer is a sum over all the tree outputs:

$$\mathcal{T}(x) = T^{(1)}(x) + T^{(2)}(x) + \cdots + T^{(m)}(x). \quad (1)$$

The output of the layer $\mathcal{T}(x)$ is a logit vector in $\mathbb{R}^k$. In the case of classification, mapping from the logit space to $\mathcal{Y}$ can be done by applying a softmax and returning the class with the highest probability. Next, we introduce the key building block of the approach: the differentiable decision tree.

The Differentiable Decision Tree: Classical trees perform *hard routing*, i.e., a sample is routed to exactly one direction at every internal node. Hard routing introduces discontinuities in the loss function, making trees unamenable to continuous optimization. Therefore, trees are usually built in a greedy fashion. In this section, we present an enhancement of the soft trees proposed by (Jordan & Jacobs, 1994) and utilized in (Kontschieder et al., 2015; Frosst & Hinton, 2017; Hehn et al., 2019). Soft trees are a variant of decision trees that perform *soft routing*, where every internal node can route the sample to the left and right simultaneously with different proportions. This routing mechanism makes soft trees differentiable, so learning can be done using gradient-based methods. Soft trees cannot route a sample only to the left or to the right, making conditional computation impossible. Subsequently, we introduce a new activation function for soft trees, which allows conditional computation while preserving differentiability.

We consider a single tree in the additive model (1), and denote the tree by T (we drop the superscript to simplify the notation). Recall that T takes an input sample and returns an output vector (logit), i.e., $T : \mathcal{X} \subseteq \mathbb{R}^p \to \mathbb{R}^k$. Moreover, we assume that T is a perfect binary tree with depth d. We use the sets $\mathcal{I}$ and $\mathcal{L}$ to denote the internal (split) nodes and the leaves of the tree, respectively. For any node $i \in \mathcal{I} \cup \mathcal{L}$, we define $\mathcal{A}(i)$ as its set of ancestors and use the notation $\{x \rightsquigarrow i\}$ for the event that a sample $x \in \mathbb{R}^p$ reaches i. A summary of all the notation used in this paper is in Table A.1 in the appendix.

Soft Routing: Internal tree nodes perform soft routing, where a sample is routed left and right with different proportions. We will introduce soft routing using a probabilistic model. While we use probability to model the routing process, we will see that the final prediction of the tree is an expectation over the leaves, making T a deterministic function. Each internal node $i \in \mathcal{I}$ is associated with a trainable weight vector $w_i \in \mathbb{R}^p$, which can be viewed as hyperplane split controlling the routing. Let $\mathcal{S} : \mathbb{R} \to [0, 1]$ be an activation function. Given a sample $x \in \mathbb{R}^p$, the probability that internal node i routes x to the left is defined by $\mathcal{S}(\langle w_i, x\rangle)$.

Now we discuss how to model the probability that x reaches a certain leaf l. Let $[l \swarrow i]$ (resp. $[i \searrow l]$) denote the event that leaf l belongs to the left (resp. right) subtree of node $i \in \mathcal{I}$. Assuming that the routing decision made at each internal node in the tree is independent of the other nodes, the probability that x reaches l is given by:

$$P(\{x \rightsquigarrow l\}) = \prod\nolimits_{i \in \mathcal{A}(l)} r_l(x; w_i), \quad (2)$$

where $r_l(x; w_i)$ is the probability of node i routing x towards the subtree containing leaf l, i.e., $r_l(x; w_i) := \mathcal{S}(\langle x, w_i\rangle)^{\mathbb{1}[l \swarrow i]}(1 - \mathcal{S}(\langle x, w_i\rangle))^{\mathbb{1}[i \searrow l]}$. Next, we define how the the root-to-leaf probabilities in (2) can be used to make the final prediction of the tree.

Prediction: As with traditional decision trees, we assume that each leaf stores a weight vector $o_l \in \mathbb{R}^k$ (learned during training). Note that, during a forward pass, o_l is a constant vector, meaning that it is not a function of the input sample(s). For a sample $x \in \mathbb{R}^p$, we define the prediction of the tree as the expected value of the leaf outputs, i.e.,

$$T(x) = \sum\nolimits_{l \in \mathcal{L}} P(\{x \rightsquigarrow l\}) o_l \quad (3)$$

Activation Functions: In soft routing, the internal nodes use an activation function $\mathcal{S}$ in order to compute the routing probabilities. The logistic (a.k.a. sigmoid) function is the common choice for $\mathcal{S}$ in the literature on soft trees (see (Jordan & Jacobs, 1994; Kontschieder et al., 2015; Frosst & Hinton, 2017; Tanno et al., 2018; Hehn et al., 2019)). While the logistic function can output arbitrarily small values, it cannot output an exact zero. This implies that any sample x will reach every node in the tree with a positive probability (as evident from (2)). Thus, computing the output of the tree in (3) will require computation over every node in the tree, an operation which is exponential in tree depth.

We propose a novel *smooth-step activation function*, which can output exact zeros and ones, thus allowing for true conditional computation. Our smooth-step function is S-shaped and continuously differentiable, similar to the logistic function. Let γ be a non-negative scalar parameter. The smooth-step function is a cubic polynomial in the interval $[-\gamma/2, \gamma/2]$, 0 to the left of the interval, and 1 to the right. More formally, we assume that the function takes the parametric form $\mathcal{S}(t) = at^3 + bt^2 + ct + d$ for $t \in [-\gamma/2, \gamma/2]$, where a, b, c, d are scalar parameters. We then solve for the parameters under the following continuity and differentiability constraints: (i) $\mathcal{S}(-\gamma/2) = 0$, (ii) $\mathcal{S}(\gamma/2) = 1$, (iii) $\mathcal{S}'(t)|_{t=-\gamma/2} = \mathcal{S}'(t)|_{t=\gamma/2} = 0$. This leads to:

$$\mathcal{S}(t) = \begin{cases} 0 & \text{if } t \leq -\gamma/2 \\ -\frac{2}{\gamma^3}t^3 + \frac{3}{2\gamma}t + \frac{1}{2} & \text{if } -\gamma/2 \leq t \leq \gamma/2 \\ 1 & \text{if } t \geq \gamma/2 \end{cases} \quad (4)$$

By construction, the smooth-step function in (4) is continuously differentiable for any $t \in \mathbb{R}$ (including $-\gamma/2$ and

$\gamma/2$). In Figure 1 (left), we plot the smooth-step (with $\gamma = 1$) and logistic activation functions; the logistic function here takes the form $(1 + e^{-6t})^{-1}$, i.e., it is a rescaled variant of the standard logistic function, so that the two functions are on similar scales. The two functions can be very close in the middle of the fractional region. The main difference is that the smooth-step function outputs exact zero and one, whereas the logistic function converges to these asymptotically. Outside $[-\gamma/2, \gamma/2]$, the smooth-step function performs hard routing, similar to classical decision trees. The choice of γ controls the fraction of samples that are hard routed. A very small γ can lead to many zero gradients in the internal nodes, whereas a very large γ might limit the extent of conditional computation. In our experiments, we use batch normalization (Ioffe & Szegedy, 2015) before the tree layer so that the inputs to the smooth-step function remain centered and bounded. This turns out to be very effective in preventing the internal nodes from having zero gradients, at least in the first few training epochs. Moreover, we view γ as a hyperparameter, which we tune over the range $[10^{-4}, 1]$. This range works well for balancing the training performance and conditional computation across the 26 datasets we used (see Section 4).

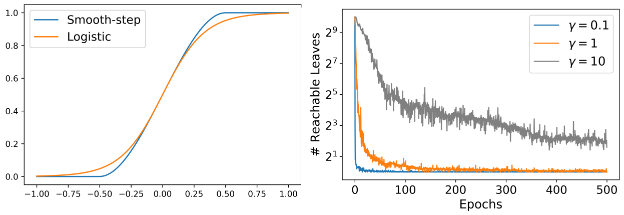

Figure 1: **Left**: Smooth-step vs. Logistic $(1 + e^{-6t})^{-1}$. **Right**: Number of reachable leaves (per sample) during training on a tree of depth 10.

For a given sample x, we say that a node i is reachable if $P(x \rightsquigarrow i) > 0$. The number of reachable leaves directly controls the extent of conditional computation. In Figure 1 (right), we plot the average number of reachable leaves (per sample) as a function of the training epochs, for a single tree of depth 10 (i.e., with 1024 leaves) and different γ's. This is for the diabetes dataset (Olson et al., 2017), using Adam (Kingma & Ba, 2014) for optimization (see the appendix for details). The figure shows that for small enough γ (e.g., $\gamma \leq 1$), the number of reachable leaves rapidly converges to 1 during training (note that the y-axis is on a log scale). We observed this behavior on all the datasets in our experiments. We note that variants of the smooth-step function are used in computer graphics in tasks such as texturing (Ebert et al., 2003). However, to our knowledge, such functions have not been used in soft trees or NNs.

In the next section, we show how the sparsity in the smooth-step function and in its gradient can be exploited to develop efficient forward and backward propagation algorithms.

3. Conditional Computation

We propose using first order optimization methods (e.g., SGD and its variants) to optimize TEL. A main computational bottleneck in this case is the gradient computation, whose time and memory complexities can grow exponentially in the tree depth. This has hindered training large tree ensembles in the literature. In this section, we develop efficient forward and backward propagation algorithms for TEL by exploiting the sparsity in both the smooth-step function and its gradient. We show that our algorithms have optimal time complexity and discuss cases where they run significantly faster than standard backpropagation.

Setup: We assume a general setting where TEL is a hidden layer. Without loss of generality, we consider only one sample and one tree. Let $x \in \mathbb{R}^p$ be the input to TEL and denote the tree output by $T(x) \in \mathbb{R}^k$, where $T(x)$ is defined in (3). We assume that a backpropagation algorithm has already computed the gradients associated with the layers after TEL and has computed $\frac{\partial L}{\partial T}$. We use the same notation as in Section 2, and we collect the leaf vectors o_l, $l \in \mathcal{L}$ into the matrix $O \in \mathbb{R}^{|\mathcal{L}| \times k}$ and the internal node weights w_i, $i \in \mathcal{I}$ into the matrix $W \in \mathbb{R}^{|\mathcal{I}| \times p}$. Moreover, for a differentiable function $h(z)$ which maps $\mathbb{R}^s \to \mathbb{R}^u$, we denote its Jacobian by $\frac{\partial h}{\partial z} \in \mathbb{R}^{u \times s}$. Let L be the loss function to be optimized (e.g., cross-entropy). Our goal is to efficiently compute the following three gradients: $\frac{\partial L}{\partial O}$, $\frac{\partial L}{\partial W}$, and $\frac{\partial L}{\partial x}$. The first two gradients are needed by the optimizer to update O and W. The third gradient is used to continue the backpropagation in the layers preceding TEL.

Number of Reachable Nodes: To exploit conditional computation effectively, each sample should reach a relatively small number of leaves. This can be enforced by choosing the parameter γ of the smooth-step function to be sufficiently small. When analyzing the complexity of the forward and backward passes below, we will assume that the sample x reaches U leaves and N internal nodes.

3.1. Conditional Forward Pass

Prior to computing the gradients, a forward pass over the tree is required. This entails computing expression (3), which is a sum of probabilities over all the root-to-leaf paths in T. Our algorithm exploits the following observation: if a certain edge on the path to leaf l has a zero probability, then $P(x \rightsquigarrow l) = 0$ so there is no need to continue evaluation along that path. Thus, we traverse the tree starting from the root, and every time a node outputs a 0 probability on one side, we ignore all of its descendants lying on that side. The summation in (3) is then performed only over the leaves reached by the traversal. We present the conditional forward pass in Algorithm 1, where for any internal node i, we denote the left and right children by $left(i)$ and $right(i)$.

Algorithm 1 Conditional Forward Pass

```
Input: Sample $x \in \mathbb{R}^p$ and tree parameters $W$ and $O$.
Output: $T(x)$
{For any node $i$, $i.prob$ denotes $P(x \rightsquigarrow i)$.}
{$to\_traverse$ is a stack for traversing nodes.}
$output \leftarrow 0$, $to\_traverse \leftarrow \{root\}$, $root.prob \leftarrow 1$
while $to\_traverse$ is not empty do
    Remove a node $i$ from $to\_traverse$
    if $i$ is an internal node then
        $left(i).prob = i.prob * \mathcal{S}(\langle w_i, x \rangle)$
        $right(i).prob = i.prob * (1 - \mathcal{S}(\langle w_i, x \rangle))$
        if $\mathcal{S}(\langle w_i, x \rangle) > 0$, add $left(i)$ to $to\_traverse$
        if $\mathcal{S}(\langle w_i, x \rangle) < 1$, add $right(i)$ to $to\_traverse$
    else
        $output \leftarrow output + i.prob * o_i$
    end if
end while
```

Time Complexity: The algorithm visits each reachable node in the tree once. Every reachable internal node requires $\mathcal{O}(p)$ operations to compute $\mathcal{S}(\langle w_i, x \rangle)$, whereas each reachable leaf requires $\mathcal{O}(k)$ operations to update the output variable. Thus, the overall complexity is $\mathcal{O}(Np + Uk)$ (recall that N and U are the number of reachable internal nodes and leaves, respectively). This is in contrast to a dense forward pass[2], whose complexity is $\mathcal{O}(2^d p + 2^d k)$ (recall that d is the depth). As long as γ is chosen so that U is sub-exponential[3] in d, the conditional forward pass has a better complexity than the dense pass (this holds since $N = \mathcal{O}(Ud)$, implying that N is also sub-exponential in d).

Memory Complexity: The memory complexity for inference and training is $\mathcal{O}(d)$ and $\mathcal{O}(d + U)$, respectively. See the appendix for a detailed analysis. This is in contrast to a dense forward pass, whose complexity in training is $\mathcal{O}(2^{\text{d}})$.

3.2. Conditional Backward Pass

Here we develop a backward pass algorithm to efficiently compute the three gradients: $\frac{\partial L}{\partial O}$, $\frac{\partial L}{\partial W}$, and $\frac{\partial L}{\partial x}$, assuming that $\frac{\partial L}{\partial T}$ is available from a backpropagation algorithm. In what follows, we will see that as long as U is sufficiently small, the gradients $\frac{\partial L}{\partial O}$ and $\frac{\partial L}{\partial W}$ will be sparse, and $\frac{\partial L}{\partial x}$ can be computed by considering only a small number of nodes in the tree. Let $\mathcal{R}$ be the set of leaves reached by Algorithm 1. The following set turns out to be critical in understanding the sparsity structure in the problem: $\mathcal{F} := \{i \in \mathcal{I} \mid i \in \mathcal{A}(l),\ l \in \mathcal{R},\ 0 < \mathcal{S}(\langle x, w_i \rangle) < 1\}$. In words, $\mathcal{F}$ is the set of ancestors of the reachable leaves, whose activation is fractional.

In Theorem 1, we show how the three gradients can be computed by only considering the internal nodes in $\mathcal{F}$ and leaves in $\mathcal{R}$. Moreover, the theorem presents sufficient conditions for which the gradients are zero; in particular, $\frac{\partial L}{\partial w_i} = 0$ for every internal node $i \in \mathcal{F}^c$ and $\frac{\partial L}{\partial o_l} = 0$ for every leaf $l \in \mathcal{R}^c$ (where A^c is the complement of a set A).

Theorem 1. *Define* $\mu_1(x, i) = \frac{\partial \mathcal{S}(\langle x, w_i \rangle)}{\partial \langle x, w_i \rangle} / \mathcal{S}(\langle x, w_i \rangle)$, $\mu_2(x, i) = \frac{\partial \mathcal{S}(\langle x, w_i \rangle)}{\partial \langle x, w_i \rangle} / (1 - \mathcal{S}(\langle x, w_i \rangle))$, *and* $g(l) = P(\{x \rightsquigarrow l\}) \langle \frac{\partial L}{\partial T}, o_l \rangle$. *The gradients needed for backpropagation can be expressed as follows:*

$$\frac{\partial L}{\partial x} = \sum_{i \in \mathcal{F}} w_i^T \Big[\mu_1(x, i) \sum_{l \in \mathcal{R} | [l \swarrow i]} g(l) - \mu_2(x, i) \sum_{l \in \mathcal{R} | [i \searrow l]} g(l) \Big]$$

$$\frac{\partial L}{\partial w_i} = \begin{cases} 0 & i \in \mathcal{F}^c \\ x^T \Big[\mu_1(x, i) \sum_{l \in \mathcal{R} | [l \swarrow i]} g(l) - \mu_2(x, i) \sum_{l \in \mathcal{R} | [i \searrow l]} g(l) \Big] & o.w. \end{cases}$$

$$\frac{\partial L}{\partial o_l} = \frac{\partial L}{\partial T} P(\{x \rightsquigarrow l\}),\ \forall\, l \in \mathcal{L}$$

In Theorem 1, the quantities $\mu_1(x, i)$ and $\mu_2(x, i)$ can be obtained in $\mathcal{O}(1)$ since in Algorithm 1 we store $\langle x, w_i \rangle$ for every $i \in \mathcal{F}$. Moreover, $P(\{x \rightsquigarrow l\})$ is stored in Algorithm 1 for every reachable leaf. However, a direct evaluation of these gradients leads to a suboptimal time complexity because the terms $\sum_{l \in \mathcal{R} | [l \swarrow i]} g(l)$ and $\sum_{l \in \mathcal{R} | [i \searrow l]} g(l)$ will be computed from scratch for every node $i \in \mathcal{F}$. Our conditional backward pass traverses a *fractional tree*, composed of only the nodes in $\mathcal{F}$ and $\mathcal{R}$, while deploying smart bookkeeping to compute these sums during the traversal and avoid recomputation. We define the fractional tree below.

Definition 1. *Let $T_{reachable}$ be the tree traversed by the conditional forward pass (Algorithm 1). We define the fractional tree $T_{fractional}$ as the result of the following two operations: (i) remove every internal node $i \in \mathcal{F}^c$ from $T_{reachable}$ and (ii) connect every node with no parent to its closest ancestor.*

In Figure C.1 in the appendix, we provide an example of how the fractional tree is constructed. $T_{\text{fractional}}$ is a binary tree with U leaves and $|\mathcal{F}|$ internal nodes, each of degree exactly 2. It can be readily seen that $|\mathcal{F}| = U - 1$; this relation is useful for analyzing the complexity of the conditional backward pass. Note that $T_{\text{fractional}}$ can be constructed on-the-fly while performing the conditional forward pass (without affecting its complexity). In Algorithm 2, we present the conditional backward pass, which traverses the fractional tree once and returns $\frac{\partial L}{\partial x}$ and any (potentially) non-zero entries in $\frac{\partial L}{\partial O}$ and $\frac{\partial L}{\partial W}$.

Time Complexity: The worst-case complexity of the algorithm is $\mathcal{O}(Up + Uk)$, whereas the best-case complexity is $\mathcal{O}(k)$ (corresponds to $U = 1$), and in the worst case, the number of non-zero entries in the three gradients is $\mathcal{O}(Up + Uk)$—see the appendix for analysis. Thus, the complexity is optimal, in the sense that it matches the number

[2] By dense forward pass, we mean evaluating the tree without conditional computation (as in a standard forward pass).

[3] A function $f(t)$ is sub-exp. in t if $\lim_{t \to \infty} \log(f(t))/t = 0$.

Algorithm 2 Conditional Backward Pass

```
Input: Sample $x \in \mathbb{R}^p$, tree parameters, and $\frac{\partial L}{\partial T}$.
Output: $\frac{\partial L}{\partial x}$ and (potential) non-zeros in $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial O}$.
$\frac{\partial L}{\partial x} = 0$
{For any node $i$, $i.sum\_g$ denotes $\sum_{l \in \mathcal{R} | i \in \mathcal{A}(l)} g(l)$}
Traverse $T_{\text{fractional}}$ in post order:
  Denote the current node by $i$
  if $i$ is a leaf then
    $\frac{\partial L}{\partial o_i} = \frac{\partial L}{\partial T} P(\{x \rightsquigarrow i\})$
    $i.sum\_g = g(i)$
  else
    $a = \mu_1(x, i)\ (left(i).sum\_g)$
    $b = \mu_2(x, i)\ (right(i).sum\_g)$
    $\frac{\partial L}{\partial x} \mathrel{+}= w_i^T(a - b)$
    $\frac{\partial L}{\partial w_i} = x^T(a - b)$
    $i.sum\_g = left(i).sum\_g + right(i).sum\_g$
  end if
```

of non-zero gradient entries, in the worst case. The worst-case complexity is generally lower than the $\mathcal{O}(Np + Uk)$ complexity of the conditional forward pass. This is because we always have $U = \mathcal{O}(N)$, and there can be many cases where N grows faster than U. For example, consider a tree with only two reachable leaves ($U = 2$) and where the root is the (only) fractional node, then N grows linearly with the depth d. As long as U is sub-exponential in d, Algorithm 2's complexity can be significantly lower than that of a dense backward pass whose complexity is $\mathcal{O}(2^d p + 2^d k)$.

Memory Complexity: We store one scalar per node in the fractional tree (i.e., $i.sum\_g$ for every node i in the fractional tree). Thus, the memory complexity is $\mathcal{O}(|\mathcal{F}| + U) = \mathcal{O}(U)$. If γ is chosen so that U is upper-bounded by a constant, then Algorithm 2 will require constant memory.

Connections to Backpropagation: An interesting observation in our approach is that the conditional backward pass generally has a better time complexity than the conditional forward pass. This is usually impossible in standard backpropagation for NNs, as the forward and backward passes traverse the same computational graph (Goodfellow et al., 2016). The improvement in complexity of the backward pass in our case is due to Algorithm 2 operating on the fractional tree, which can contain a significantly smaller number of nodes than the tree traversed by the forward pass. In the language of backpropagation, our fractional tree can be viewed as a "simplified" computational graph, where the simplifications are due to Theorem 1.

4. Experiments

We study the performance of TEL in terms of prediction, conditional computation, and compactness. We evaluate TEL as a standalone learner and as a layer in a NN, and compare to standard soft trees, GBDT, and dense layers.

Model Implementation: TEL is implemented in TensorFlow 2.0 using custom C++ kernels for forward and backward propagation, and is open-sourced. Keras Python-accessible interface is also available.

Datasets: We use a collection of 26 classification datasets (binary and multiclass) from various domains (e.g., healthcare, genetics, and image recognition). 23 of these are from the Penn Machine Learning Benchmarks (PMLB) (Olson et al., 2017) and the 3 remaining are CIFAR-10 (Krizhevsky et al., 2009), MNIST (LeCun et al., 1998), and Fashion MNIST (Xiao et al., 2017). Details are in the appendix.

Tuning, Toolkits, and Details: For all the experiments, we tune the key hyperparameters using Hyperopt (Bergstra et al., 2013) with the Tree-structured Parzen Estimator (TPE). We optimize for either AUC or accuracy with stratified 5-fold CV. NNs (including TEL) were trained using Keras with the TensorFlow backend, using Adam (Kingma & Ba, 2014) and cross-entropy loss. As discussed in Section 2, TEL is always preceded by a batch normalization layer. GBDT are from XGBoost (Chen & Guestrin, 2016), Logistic regression and CART are from Scikit-learn (Pedregosa et al., 2011). Additional details are in the appendix.

4.1. Soft Trees: Smooth-step vs. Logistic Activation

We compare the run time and performance of the smooth-step and logistic functions using 23 PMLB datasets.

Predictive Performance: We fix the TEL architecture to 10 trees of depth 4. We tune the learning rate, batch size, and number of epochs (ranges are in the appendix). We assume the following parametric form for the logistic function $f(t) = (1 + e^{-t/\alpha})^{-1}$, where α is a hyperparameter which we tune in the range $[10^{-4}, 10^4]$. The smooth-step's parameter γ is tuned in the range $[10^{-4}, 1]$. Here we restrict the upper range of γ to 1 to enable conditional computation over the whole tuning range. While γ's larger than 1 can lead to slightly better predictive performance in some cases, they can slow down training significantly. For tuning, Hyperopt is run for 50 rounds with AUC as the metric. After tuning, models with the best hyperparameters are retrained. We repeat the training procedure 5 times using random weight initializations. The mean test AUC along with its standard error (SE) are in Table 2. The smooth-step outperforms the logistic function on 7 datasets (5 are statistically significant). The logistic function also wins on 7 datasets (4 are statistically significant). The two functions match on the rest of the datasets. The differences on the majority of the datasets are small (even when statistically significant), suggesting that using the smooth-step function does not hurt the predictive performance. However, as we will see next, the smooth-step has a significant edge in terms of computation time.

Training Time: We measure the training time over 50 epochs as a function of tree depth for both activation func-

Table 2: Test AUC for the smooth-step and logistic functions (fixed TEL architecture). A $*$ indicates statistical significance based on a paired two-sided t-test at a significance level of 0.05. Best results are in **bold**. AUCs on the 9 remaining datasets match and are hence omitted.

Dataset	Smooth-step	Logistic
ann-thyroid	$\mathbf{0.997} \pm 0.0001$	0.996 ± 0.0006
breast-cancer-w.	0.992 ± 0.0015	$\mathbf{0.994} \pm 0.0002$
churn	0.897 ± 0.0014	$\mathbf{0.898} \pm 0.0014$
crx	0.916 ± 0.0025	$\mathbf{0.929}^* \pm 0.0021$
diabetes	$\mathbf{0.832}^* \pm 0.0009$	0.816 ± 0.0021
dna	0.993 ± 0.0004	$\mathbf{0.994}^* \pm 0.0$
ecoli	$\mathbf{0.97}^* \pm 0.0004$	0.952 ± 0.0038
flare	0.78 ± 0.0027	$\mathbf{0.784} \pm 0.0018$
heart-c	$\mathbf{0.936} \pm 0.002$	0.927 ± 0.0036
pima	$\mathbf{0.828}^* \pm 0.0005$	0.82 ± 0.0003
satimage	$\mathbf{0.988}^* \pm 0.0002$	0.987 ± 0.0002
solar-flare_2	0.926 ± 0.0002	$\mathbf{0.927}^* \pm 0.0007$
vehicle	0.956 ± 0.0015	$\mathbf{0.965}^* \pm 0.0007$
yeast	$\mathbf{0.876}^* \pm 0.0014$	0.86 ± 0.0026

tions. We keep the same ensemble size (10) and use $\gamma = 1$ for the smooth-step as this corresponds to the worst-case training time (in the tuning range $[10^{-4}, 1]$), and we fix the optimization hyperparameters (batch size = 256 and learning rate = 0.1). We report the results for three of the datasets in Figure 2; the results for the other datasets have very similar trends and are omitted due to space constraints. The results indicate a steep exponential increase in training time for the logistic activation after depth 6. In contrast, the smooth-step has a slow growth, achieving over 10x speed-up at depth 10.

4.2. TEL vs. Gradient Boosted Decision Trees

Predictive Performance: We compare the predictive performance of TEL and GBDT on the 23 PMLB datasets, and we include L2-regularized logistic regression (LR) and CART as baselines. For a fair comparison, we use TEL as a standalone layer. For TEL and GBDT, we tune over the # of trees, depth, learning rate, and L2 regularization. For TEL we also tune over the batch size, epochs, and $\gamma \in [10^{-4}, 1]$. For LR and CART, we tune the L2 regularization and depth, respectively. We use 50 tuning rounds in Hyperopt with the AUC metric. We repeat the tuning/testing procedures on 15 random training/testing splits. Results are in Table 3.

As expected, no algorithm dominates on all the datasets. TEL outperforms GBDT on 9 datasets (5 are statistically significant). GBDT outperforms TEL on 8 datasets (7 of which are statistically significant). There were ties on the 6 remaining datasets; these typically correspond to easy tasks where an AUC of (almost) 1 can be attained. LR outperforms both TEL and GBDT on only 3 datasets with very marginal difference. Overall, the results indicate that TEL's performance is competitive with GBDT. Moreover, adding feature representation layers before TEL can potentially improve its performance further, e.g., see Sec. 4.3.

Compactness and Sensitivity: We compare the number of trees and sensitivity of TEL and GBDT on datasets from Table 3 where both models achieve comparable AUCs—namely, the heart-c, pima and spambase datasets. With similar predictive performance, compactness can be an important factor in choosing a model over the other. For TEL, we use the models trained in Table 3. As for GBDT, for each dataset, we fix the depth so that the number of parameters per tree in GBDT (roughly) matches that of TEL. We tune over the main parameters of GBDT (50 iterations of Hyperopt, under the same parameter ranges of Table 3). We plot the test AUC versus the number of trees in Figure 3. On all datasets, the test AUC of TEL peaks at a significantly smaller number of trees compared to GBDT. For example, on pima, TEL's AUC peaks at 5 trees, whereas GBDT requires more than 100 trees to achieve a comparable performance—this is more than 20x reduction in the number of parameters. Moreover, the performance of TEL is less sensitive w.r.t. to changes in the number of trees. These observations can be attributed to the joint optimization performed in TEL, which can lead to more expressive ensembles compared to the stage-wise optimization in GBDT.

4.3. TEL vs. Dense Layers in CNNs

We study the benefits of replacing dense layers with TEL in CNNs, on the CIFAR-10, MNIST, and Fashion MNIST datasets. We consider 2 convolutional layers, followed by intermediate layers (max pooling, dropout, batch normalization), and finally dense layers (we will refer to this as CNN-dense). We also consider a similar architecture, where we replace the final dense layers with a single dense layer followed by TEL (CNN-TEL model). We tune over the optimization hyperparameters, the number of filters in the convolutional layers, the number and width of the dense layers, and the different parameters of TEL (see appendix for details). We run Hyperopt for 25 iterations with classification accuracy as the target metric. After tuning, the models are trained using 5 random weight initializations. The classification accuracy and loss on the test set and the total number of parameters are reported in Table 4. While the accuracies are comparable, CNN-TEL achieves a lower test loss on the three datasets, where the 28% and 53% relative improvements on CIFAR and Fashion MNIST are statistically significant. Since we are using cross-entropy loss, this means that TEL gives higher scores on average, when it makes correct predictions. Moreover, the number of parameters in CNN-TEL is $\sim$ 8x smaller than CNN-dense. This example also demonstrates how representation layers can be effectively leveraged by TEL—GBDT's performance is significantly lower on the MNIST and CIFAR-10 (e.g., see the comparisons in (Ponomareva et al., 2017)).

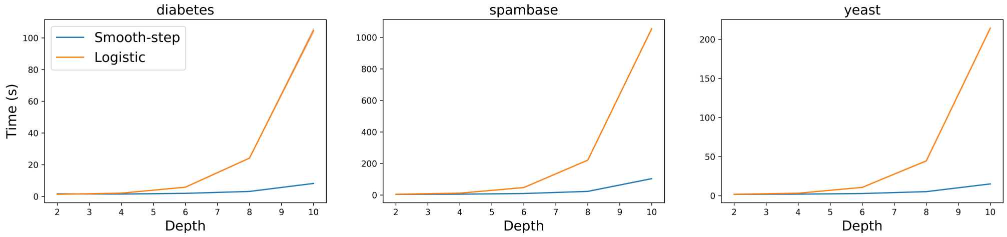

Figure 2: Training time (sec) vs tree depth for the smooth-step and logistic functions, averaged over 5 repetitions.

Table 3: Test AUC on 23 PMLB datasets. Averages over 15 random repetitions are reported along with the SE. A star$*$ indicates statistical significance based on a paired two-sided t-test at a significance level of 0.05. Best results are in **bold**.

Dataset	TEL	GBDT	L2 Logistic Reg.	CART
ann-thyroid	0.996 ± 0.0	$\mathbf{1.0}^{*} \pm 0.0$	0.92 ± 0.002	0.997 ± 0.0
breast-cancer-wisconsin	$\mathbf{0.995}^{*} \pm 0.001$	0.992 ± 0.001	0.991 ± 0.001	0.929 ± 0.004
car-evaluation	$\mathbf{1.0} \pm 0.0$	$\mathbf{1.0} \pm 0.0$	0.985 ± 0.001	0.981 ± 0.001
churn	0.916 ± 0.004	$\mathbf{0.92}^{*} \pm 0.004$	0.814 ± 0.003	0.885 ± 0.004
crx	0.911 ± 0.005	$\mathbf{0.933}^{*} \pm 0.004$	0.916 ± 0.005	0.905 ± 0.005
dermatology	$\mathbf{0.998} \pm 0.001$	$\mathbf{0.998} \pm 0.001$	$\mathbf{0.998} \pm 0.001$	0.962 ± 0.005
diabetes	$\mathbf{0.831}^{*} \pm 0.006$	0.82 ± 0.006	0.824 ± 0.008	0.774 ± 0.008
dna	0.993 ± 0.0	$\mathbf{0.994}^{*} \pm 0.0$	0.991 ± 0.0	0.964 ± 0.001
ecoli	$0.97^{*} \pm 0.003$	0.962 ± 0.003	$\mathbf{0.972} \pm 0.003$	0.902 ± 0.007
flare	0.732 ± 0.009	$\mathbf{0.738} \pm 0.01$	0.736 ± 0.009	0.717 ± 0.01
heart-c	0.903 ± 0.006	0.893 ± 0.008	$\mathbf{0.908} \pm 0.005$	0.829 ± 0.012
hypothyroid	0.971 ± 0.003	$\mathbf{0.987}^{*} \pm 0.002$	0.93 ± 0.005	0.926 ± 0.011
nursery	$\mathbf{1.0} \pm 0.0$	$\mathbf{1.0} \pm 0.0$	0.916 ± 0.001	0.996 ± 0.0
optdigits	$\mathbf{1.0} \pm 0.0$	$\mathbf{1.0} \pm 0.0$	0.998 ± 0.0	0.958 ± 0.001
pima	0.831 ± 0.008	0.825 ± 0.006	$\mathbf{0.832} \pm 0.008$	0.758 ± 0.011
satimage	$\mathbf{0.99} \pm 0.0$	$\mathbf{0.99} \pm 0.0$	0.955 ± 0.001	0.949 ± 0.001
sleep	0.925 ± 0.0	$\mathbf{0.927}^{*} \pm 0.0$	0.889 ± 0.0	0.876 ± 0.001
solar-flare_2	$\mathbf{0.925} \pm 0.002$	0.924 ± 0.002	0.92 ± 0.002	0.907 ± 0.002
spambase	0.986 ± 0.001	$\mathbf{0.989}^{*} \pm 0.001$	0.972 ± 0.001	0.926 ± 0.002
texture	$\mathbf{1.0} \pm 0.0$	$\mathbf{1.0} \pm 0.0$	$\mathbf{1.0} \pm 0.0$	0.974 ± 0.001
twonorm	$\mathbf{0.998}^{*} \pm 0.0$	0.997 ± 0.0	$\mathbf{0.998} \pm 0.0$	0.865 ± 0.002
vehicle	$\mathbf{0.953}^{*} \pm 0.003$	0.931 ± 0.002	0.941 ± 0.002	0.871 ± 0.004
yeast	$\mathbf{0.861} \pm 0.004$	0.859 ± 0.004	0.852 ± 0.004	0.779 ± 0.005

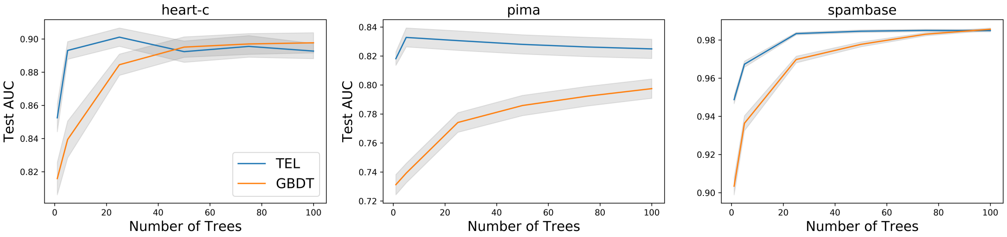

Figure 3: Mean test AUC vs # of trees (15 trials). SE is shaded. TEL and GBDT have (roughly) the same # of params/tree.

Table 4: Average and SE for test accuracy, loss and # of params for CNN-Dense and CNN-TEL over 5 random initializations. A star $*$ indicates statistical significance based on a paired two-sided t-test at a level of 5%. Best values are in **bold**.

	CNN-Dense			CNN-TEL		
Dataset	Accuracy	Loss	# Params	Accuracy	Loss	# Params
CIFAR10	0.7278 ± 0.0047	1.673 ± 0.170	$7,548,362$	$\mathbf{0.7296} \pm 0.0109$	$\mathbf{1.202}^{*} \pm 0.011$	$\mathbf{926,465}$
MNIST	0.9926 ± 0.0002	0.03620 ± 0.00121	$5,830,538$	$\mathbf{0.9930} \pm 9e-5$	$\mathbf{0.03379} \pm 0.00093$	$\mathbf{699,585}$
Fashion MNIST	$\mathbf{0.9299} \pm 0.0012$	0.6930 ± 0.0291	$5,567,882$	0.9297 ± 0.0012	$\mathbf{0.3247}^{*} \pm 0.0045$	$\mathbf{699,585}$

Acknowledgments: We would like to thank Mehryar Mohri for the useful discussions. Part of this work was done when Hussein was at Google Research. At MIT, Hussein acknowledges research funding from the Office of Naval Research onr-n000141812298. Rahul acknowledges research funding from the Office of Naval Research onr-n000141812298 (Young Investigator Award), the National Science Foundation (NSF-IIS-1718258), and IBM.

References

Bengio, E., Bacon, P., Pineau, J., and Precup, D. Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297, 2015. URL http://arxiv.org/abs/1511.06297.

Bengio, Y., Courville, A., and Vincent, P. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8): 1798–1828, 2013.

Bergstra, J., Yamins, D., and Cox, D. D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML13, pp. I115I123. JMLR.org, 2013.

Biau, G., Scornet, E., and Welbl, J. Neural random forests. *Sankhya A*, 81(2):347–386, Dec 2019. ISSN 0976-8378. doi: 10.1007/s13171-018-0133-y. URL https://doi.org/10.1007/s13171-018-0133-y.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. Classification and regression trees. 1983.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

Friedman, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pp. 1189–1232, 2001.

Frosst, N. and Hinton, G. E. Distilling a neural network into a soft decision tree. In *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML 2017*, CEUR Workshop Proceedings, 2017.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Hastie, T., Tibshirani, R., and Friedman, J. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Hehn, T. M., Kooij, J. F., and Hamprecht, F. A. End-to-end learning of decision trees and forests. *International Journal of Computer Vision*, pp. 1–15, 2019.

Ioannou, Y., Robertson, D., Zikic, D., Kontschieder, P., Shotton, J., Brown, M., and Criminisi, A. Decision forests, convolutional networks and the models in-between, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML15, pp. 448456. JMLR.org, 2015.

Jernite, Y., Choromanska, A., and Sontag, D. Simultaneous learning of trees and representations for extreme classification and density estimation. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1665–1674, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/jernite17a.html.

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

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kontschieder, P., Fiterau, M., Criminisi, A., and Bul, S. R. Deep neural decision forests. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1467–1475, Dec 2015. doi: 10.1109/ICCV.2015.172.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Olson, R. S., La Cava, W., Orzechowski, P., Urbanowicz, R. J., and Moore, J. H. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017. ISSN 1756-0381. doi: 10.1186/s13040-017-0154-4. URL https://doi.org/10.1186/s13040-017-0154-4.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Ponomareva, N., Colthurst, T., Hendry, G., Haykal, S., and Radpour, S. Compact multi-class boosted trees. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 47–56. IEEE, 2017.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538.

Tanno, R., Arulkumaran, K., Alexander, D. C., Criminisi, A., and Nori, A. Adaptive neural trees. *arXiv preprint arXiv:1807.06699*, 2018.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Yu, D. and Deng, L. *Automatic Speech Recognition*. Springer, 2016.

Zoran, D., Lakshminarayanan, B., and Blundell, C. Learning deep nearest neighbor representations using differentiable boundary trees. *CoRR*, abs/1702.08833, 2017. URL http://arxiv.org/abs/1702.08833.

A. Notation

Table A.1 lists the notation used throughout the paper.

B. Appendix for Section 2

Figure 1 (right) was generated by training a single tree with depth 10 using the smooth-step activation function. We optimized the cross-entropy loss using Adam (Kingma & Ba, 2014) with base learning rate $= 0.1$ and batch size $= 256$. The y-axis of the plot corresponds to the average number of reachable leaves per sample (each point in the graph corresponds to a batch, and averaging is done over the samples in the batch). For details on the diabetes dataset used in this experiment and the computing setup, please refer to Section D of the appendix.

C. Appendix for Section 3

C.1. Example of Conditional Forward and Backward Passes

Figure C.1 (see below) shows an example of the tree traversed by the conditional forward pass, along with the corresponding $T_{fractional}$ used during the backward pass, for a simple regression tree of depth $d = 4$. Note that only 3 leaves are reachable (out of the 16 leaves of a perfect, depth-4 tree). The values inside the boxes at the bottom correspond to the regression values (scalars) stored at each leaf. The output of the tree for the forward pass shown on the left of Figure C.1, is given by

$$T(x) = 0.8 \cdot 0.3 \cdot 1.5 + 0.8 \cdot 0.7 \cdot (-2.0) + 0.2 \cdot 2.1 = -0.34.$$

Note also that the tree used to compute the backward pass is substantially smaller than the forward pass tree since all the hard-routing internal nodes of the latter have been removed.

C.2. Memory Complexity of the Conditional Forward Pass

The memory requirements depend on whether the forward pass is being used for inference or training. For inference, a node can be discarded as soon as it is traversed. Since we traverse the tree in a depth-first manner, the worst-case memory complexity is $\mathcal{O}(d)$. When used in the context of training, additional quantities need to be stored in order to perform the backward pass efficiently. In particular, we need to store $l.prob$ for every reachable leaf l, and $\langle w_i, x\rangle$ for every internal node $i \in \mathcal{F}$, where $\mathcal{F}$ is set of ancestors of the reachable leaves whose activation is fractional—see Section 3.2 for a formal definition of $\mathcal{F}$. Note that $|F| = U - 1$ (as discussed after Definition 1). Thus, the worst-case memory complexity when used in the context of training is $\mathcal{O}(d+U)$.

C.3. Time Complexity of the Conditional Backward Pass

Lines 8 and 9 perform $\mathcal{O}(k)$ operations so each leaf requires $\mathcal{O}(k)$ operations. Lines 11 and 12 are $\mathcal{O}(1)$ since for every $i \in \mathcal{F}$, $\langle w_i, x\rangle$ is available from the conditional forward pass. Lines 13 and 14 are $\mathcal{O}(p)$, while line 15 is $\mathcal{O}(1)$. The total number of internal nodes traversed is $|\mathcal{F}|$. Moreover, we always have $|\mathcal{F}| = U - 1$ (see the discussion following Definition 1). Therefore, the worst-case complexity is $\mathcal{O}(Up + Uk)$. By Theorem 1, the maximum number of non-zero entries in the three gradients is $p + p|\mathcal{F}| + Uk = \mathcal{O}(Up + Uk)$ (and it is easy to see that this rate is achievable). The best-case complexity of Algorithm 2 is $\mathcal{O}(k)$—this corresponds to the case where there is only one reachable leaf ($U = 1$), so the fractional tree is composed of a single node.

C.4. Proof of Theorem 1

Gradient of Loss w.r.t. x: By the chain rule, we have:

$$\underbrace{\frac{\partial L}{\partial x}}_{1\times p} = \underbrace{\frac{\partial L}{\partial T}}_{1\times k}\underbrace{\frac{\partial T}{\partial x}}_{k\times p} \tag{5}$$

The first term in the RHS above is available from backpropagation. The second term can be written more explicitly as follows:

$$\begin{aligned}\frac{\partial T}{\partial x} &= \frac{\partial \sum_{l\in\mathcal{L}} P(\{x \rightsquigarrow l\})o_l}{\partial x} \\ &= \sum_{l\in\mathcal{L}} o_l \frac{\partial}{\partial x}\prod_{j\in\mathcal{A}(l)} r_l(x; w_j) \\ &= \sum_{l\in\mathcal{L}} o_l \sum_{i\in\mathcal{A}(l)} \frac{\partial}{\partial x} r_l(x; w_i) \prod_{j\in\mathcal{A}(l), j\neq i} r_l(x; w_j)\end{aligned}$$

We make three observations that allows us to simplify the expression above. First, if a leaf l is not reachable by x, then the inner term in the second summation above must be 0. This means that the outer summation can be restricted to the set of reachable leaves $\mathcal{R}$. Second, if an internal node i has a non-fractional $r_l(x; w_i)$, then $\frac{\partial}{\partial x} r_l(x; w_i) = 0$. This implies that we can restrict the inner summation to be only over $\mathcal{A}(l) \cap \mathcal{F}$. Third, the second term in the inner summation can be simplified by noting that the following holds for any $l \in \mathcal{R}$:

$$\prod_{j\in\mathcal{A}(l), j\neq i} r_l(x; w_j) = \frac{P(\{x \rightsquigarrow l\})}{r_l(x; w_i)} \tag{6}$$

Note that in the above, $r_l(x; w_i)$ cannot be zero since $l \in \mathcal{R}$ (otherwise, l will be unreachable). Combining the three observations above, $\frac{\partial T}{\partial x}$ simplifies to:

$$\frac{\partial T}{\partial x} = \sum_{l\in\mathcal{R}} o_l \sum_{i\in\mathcal{A}(l)\cap\mathcal{F}} \frac{\partial}{\partial x} r_l(x; w_i) \frac{P(\{x \rightsquigarrow l\})}{r_l(x; w_i)}. \tag{7}$$

Table A.1: List of notation used.

Notation	Space or Type	Explanation
$\mathcal{X}$	$\mathbb{R}^p$	Input feature space.
$\mathcal{Y}$	$\mathbb{R}^k$	Output (label) space.
m	$\mathbb{Z}_{>0}$	Number of trees in the TEL.
$\mathcal{T}(x)$	Function	The output of TEL, a function that takes an input sample and returns a logit which corresponds to the sum of all the trees in the ensemble. Formally, $\mathcal{T} : \mathcal{X} \to \mathbb{R}^k$.
$T(x)$	Function	A single perfect binary tree which takes an input sample and returns a logit, i.e., $T : \mathcal{X} \to \mathbb{R}^k$.
d	$\mathbb{Z}_{>0}$	The depth of tree T.
$\mathcal{I}$	Set	The set of internal (split) nodes in T.
$\mathcal{L}$	Set	The set of leaf nodes in T.
$\mathcal{A}(i)$	Set	The set of ancestors of node i.
$\{x \rightsquigarrow i\}$	Event	The event that sample $x \in \mathbb{R}^p$ reaches node i.
w_i	$\mathbb{R}^p$	Weight vector of internal node i (trainable). Defines the hyperplane split used in sample routing.
W	$\mathbb{R}^{\lvert\mathcal{I}\rvert \times p}$	Matrix of all the internal nodes weights.
$\mathcal{S}$	Function	Activation function $\mathbb{R} \to [0, 1]$
$\mathcal{S}(\langle w_i, x\rangle)$	$[0, 1]$	Probability (proportion) that internal node i routes x to the left.
$[l \swarrow i]$	Event	The event that leaf l belongs to the left subtree of node $i \in \mathcal{I}$.
$[l \searrow i]$	Event	The event that leaf l belongs to the right subtree of node $i \in \mathcal{I}$.
o_l	$\mathbb{R}^k$	Leaf l's weight vector (trainable).
O	$\mathbb{R}^{\lvert\mathcal{L}\rvert \times k}$	Matrix of leaf weights.
γ	$\mathbb{R}_{\geq 0}$	Non-negative scaling parameter for the smooth-step activation function.
L	Function	Loss function for training (e.g., cross-entropy).
U, N	$\mathbb{Z}_{>0}$	Number of leaves and internal nodes, respectively, that a sample x reaches.
$\mathcal{R}$	Set	The set of reachable leaves.
$\mathcal{F}$	Set	The set of ancestors of the reachable leaves, whose activation is fractional, i.e., $\mathcal{F} = \{i \in \mathcal{I} \mid i \in \mathcal{A}(l),\ l \in \mathcal{R},\ 0 < \mathcal{S}(\langle x, w_i\rangle) < 1\}$.

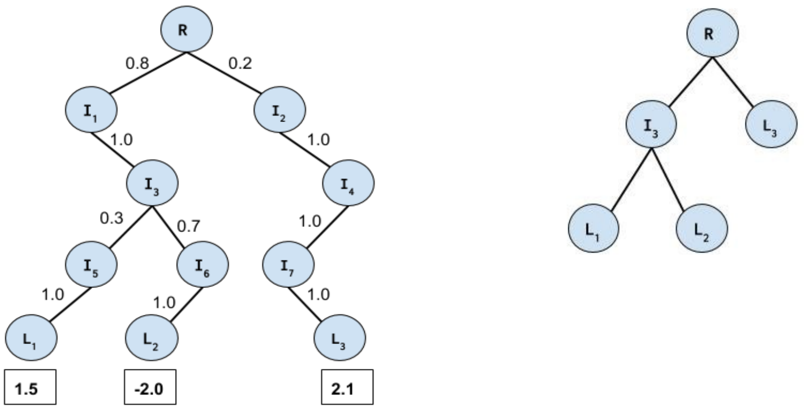

Figure C.1: **Left**: Reachable sub-tree for the conditional forward pass. **Right**: Corresponding (fractional) tree for the conditional backward pass where most of the internal (splitting) nodes of the forward pass sub-tree have been eliminated.

Note that:

$$\frac{\partial}{\partial x} r_l(x; w_i) = \mathcal{S}'(\langle x, w_i \rangle) w_i^T (-1)^{\mathbb{1}[i \searrow l]}. \tag{8}$$

Plugging (8) into (7), and then using (5), we get:

$$\frac{\partial L}{\partial x} = \sum_{l \in \mathcal{R}} g(l) \sum_{i \in \mathcal{A}(l) \cap \mathcal{F}} w_i^T (-1)^{\mathbb{1}[i \searrow l]} \frac{\mathcal{S}'(\langle x, w_i \rangle)}{r_l(x; w_i)} \tag{9}$$

where

$$g(l) = P(\{x \rightsquigarrow l\}) \langle \frac{\partial L}{\partial T}, o_l \rangle. \tag{10}$$

Finally, we switch the order of the two summations in (9) to get:

$$\begin{aligned}
\frac{\partial L}{\partial x} &= \sum_{i \in \mathcal{F}} \sum_{l \in \mathcal{R} | i \in \mathcal{A}(l)} g(l) w_i^T (-1)^{\mathbb{1}[i \searrow l]} \frac{\mathcal{S}'(\langle x, w_i \rangle)}{r_l(x; w_i)} \\
&= \sum_{i \in \mathcal{F}} \frac{\mathcal{S}'(\langle x, w_i \rangle)}{\mathcal{S}(\langle x, w_i \rangle)} w_i^T \sum_{l \in \mathcal{R} | [l \swarrow i]} g(l) \\
&\quad - \sum_{i \in \mathcal{F}} \frac{\mathcal{S}'(\langle x, w_i \rangle)}{1 - \mathcal{S}(\langle x, w_i \rangle)} w_i^T \sum_{l \in \mathcal{R} | [i \searrow l]} g(l)
\end{aligned}$$

Gradient of Loss w.r.t. w_i: By the chain rule:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial T} \frac{\partial T}{\partial w_i} \tag{11}$$

The first term in the summation is provided from backpropagation. The second term can be simplified as follows:

$$\begin{aligned}
\frac{\partial T}{\partial w_i} &= \frac{\partial \sum_{l \in \mathcal{L}} P(\{x \rightsquigarrow l\}) o_l}{\partial w_i} \\
&= \sum_{l \in \mathcal{L}} o_l \frac{\partial}{\partial w_i} \prod_{j \in \mathcal{A}(l)} r_l(x; w_j) \\
&= \sum_{l \in \mathcal{L} | i \in \mathcal{A}(l)} o_l \frac{\partial}{\partial w_i} r_l(x; w_i) \prod_{j \in \mathcal{A}(l), j \neq i} r_l(x; w_j),
\end{aligned} \tag{12}$$

If $i \in \mathcal{F}^c$ then the term inside the summation above must be zero, which leads to $\frac{\partial T}{\partial w_i} = 0$.

Next, we assume that $i \in \mathcal{F}$. If if a leaf l is not reachable, then the term inside the summation of (12) is zero. Using this observation along with (6), and simplifying we get the following for every $i \in \mathcal{F}$:

$$\begin{aligned}
\frac{\partial L}{\partial w_i} &= \frac{\mathcal{S}'(\langle x, w_i \rangle)}{\mathcal{S}(\langle x, w_i \rangle)} x^T \sum_{l \in \mathcal{R} | [l \swarrow i]} g(l) \\
&\quad - \frac{\mathcal{S}'(\langle x, w_i \rangle)}{1 - \mathcal{S}(\langle x, w_i \rangle)} x^T \sum_{l \in \mathcal{R} | [i \searrow l]} g(l)
\end{aligned}$$

Gradient of Loss w.r.t. O: Note that

$$\begin{aligned}
\frac{\partial T}{\partial o_l} &= \frac{\partial \sum_{v \in \mathcal{L}} P(\{x \rightsquigarrow v\}) o_v}{\partial o_l} \qquad (13) \\
&= P(\{x \rightsquigarrow l\}) I_k, \qquad (14)
\end{aligned}$$

where I_k is the $k \times k$ identity matrix. Applying the chain rule, we get:

$$\frac{\partial L}{\partial o_l} = \frac{\partial L}{\partial T} P(\{x \rightsquigarrow l\}). \tag{15}$$

D. Appendix for Section 4

Datasets: We consider 23 classification datasets from the Penn Machine Learning Benchmarks (PMLB) repository[4] (Olson et al., 2017). No additional preprocessing was done as the PMLB datasets are already preprocessed—see (Olson et al., 2017) for details. We randomly split each of the PMLB datasets into 70% training and 30% testing sets. The three remaining datasets are CIFAR-10 (Krizhevsky et al., 2009), MNIST (LeCun et al., 1998), and Fashion MNIST (Xiao et al., 2017). For these, we kept the original training/testing splits (60K/10K for MNIST and Fashion MNIST, and 50K/10K for CIFAR) and normalized the pixel values to the range $[0, 1]$. A summary of the 26 datasets considered is in Table D.2.

Computing Setup: We used a cluster running CentOS 7 and equipped with Intel Xeon Gold 6130 CPUs (with a 2.10GHz clock). The tuning and training was done in parallel over the competing models and datasets (i.e., each (model,dataset) pair corresponds to a separate job). For the experiments of Sections 4.1 and 4.2, each job involving TEL and XGBoost was restricted to 4 cores and 8GB of RAM, whereas LR and CART were restricted to 1 core and 2GB. The jobs in the experiment of Section 4.3 were each restricted to 8 cores and 32GB RAM. We used Python 3.6.9 to run the experiments with the following libraries: TensorFlow 2.1.0-dev20200106, XGBoost 0.90, Sklearn 0.19.0, Hyperopt 0.2.2, Numpy 1.17.4, Scipy 1.4.1, and GCC 6.2.0 (for compiling the custom forward/backward passes).

D.1. Tuning Parameters and Architectures:

A list of all the tuning parameters and their distributions is given for every experiment below. For experiment 4.3, we also describe the architectures used in detail.

Experiment of Section 4.1 For the predictive performance experiment, we use the following:

- Learning rate: Uniform over $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$.

[4] https://github.com/EpistasisLab/penn-ml-benchmarks

Table D.2: Dataset Statistics

Dataset	# samples	# features	# classes
ann-thyroid	7200	21	3
breast-cancer-w.	569	30	2
car-evaluation	1728	21	4
churn	5000	20	2
CIFAR	60000	3072	10
crx	690	15	2
dermatology	366	34	6
diabetes	768	8	2
dna	3186	180	3
ecoli	327	7	5
Fashion MNIST	70000	784	10
flare	1066	10	2
heart-c	303	13	2
hypothyroid	3163	25	2
MNIST	70000	784	10
nursery	12958	8	4
optdigits	5620	64	10
pima	768	8	2
satimage	6435	36	6
sleep	105908	13	5
solar-flare_2	1066	12	6
spambase	4601	57	2
texture	5500	40	11
twonorm	7400	20	2
vehicle	846	18	4
yeast	1479	8	9

- Batch size: Uniform over $\{32, 64, 128, 256, 512\}$.
- Number of Epochs: Discrete uniform with range $[5, 100]$.
- α: Log uniform over the range $[10^{-4}, 10^{4}]$.
- γ: Log uniform over the range $[10^{-4}, 1]$.

Experiment of Section 4.2:

TEL

- Learning rate: Uniform over $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$.
- Batch size: Uniform over $\{32, 64, 128, 256, 512\}$.
- Number of Epochs: Discrete uniform over $[5, 100]$.
- γ: Log uniform over $[10^{-4}, 1]$.
- Tree Depth: Discrete uniform over $[2, 8]$.
- Number of Trees: Discrete uniform over $[1, 100]$.
- L2 Regularization for W: Mixture model of 0 and the log uniform distribution over $[10^{-8}, 10^{2}]$. Mixture weights are 0.5 for each.

XGBoost

- Learning rate: Uniform over $\{1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$.
- Tree Depth: Discrete uniform over $[2, 20]$.
- Number of Trees: Discrete uniform over $[1, 500]$.
- L2 Regularization (λ): Mixture model of 0 and the log uniform distribution over $[10^{-8}, 10^{2}]$. Mixture weights are 0.5 for each.
- min_child_weight = 0.

Logistic Regression: We used Sklearn's default optimizer and increased the maximum number of iterations to 1000. We tuned over the L2 regularization parameter (C): Log uniform over $[10^{-8}, 10^{4}]$.

CART: We used Sklearn's "DecisionTreeClassifier" and tuned over the depth: discrete uniform over $[2, 20]$.

Experiment of Section 4.3: CNN-Dense has the following architecture:

- Convolutional layer 1: has f filters and a 3×3 kernel (where f is a tuning parameter).
- Convolutional layer 2: has $2f$ filters and a 3×3 kernel.
- 2×2 max pooling
- Flattening
- Dropout: the dropout rate is a tuning parameter.
- Batch Normalization
- Dense layers: a stack of ReLU-activated dense layers, where the number of layers and the units in each is a tuning parameter.
- Output layer: dense layer with softmax activation.

CNN-TEL has the following architecture:

- Convolutional layer 1: has f filters and a 3×3 kernel (where f is a tuning parameter).
- Convolutional layer 2: has $2f$ filters and a 3×3 kernel.
- 2×2 max pooling
- Flattening
- Dropout: the dropout rate is a tuning parameter.
- Batch Normalization
- Dense layer: the number of units is a tuning parameter.
- TEL

- Output layer: softmax.

We used the following hyperparameter distributions:

- Learning rate: Uniform over $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$.
- Batch size: Uniform over $\{32, 64, 128, 256, 512\}$.
- Number of Epochs: Discrete uniform over $[1, 100]$.
- γ: Log uniform over $[10^{-4}, 1]$.
- Tree Depth: Discrete uniform over $[2, 6]$.
- Number of Trees: Discrete uniform over $[1, 50]$.
- f: Uniform over $\{4, 8, 16, 32\}$.
- Number of dense layers in CNN-Dense: Discrete Uniform over $[1, 5]$.
- Number of units in dense layers of CNN-Dense: Uniform over $\{16, 32, 64, 128, 256, 512\}$.
- Number of units in the dense layer of CNN-TEL: Uniform over $\{16, 32, 64\}$.
- Dropout rate: Uniform over $[0.1, 0.5]$.