Softmax Tree: An Accurate, Fast Classifier When the Number of Classes Is Large

Arman Zharmagambetov **Magzhan Gabidolla** **Miguel Á. Carreira-Perpiñán**
Dept. of Computer Science and Engineering, University of California, Merced, USA
{azharmagambetov,mgabidolla,mcarreira-perpinan}@ucmerced.edu

Abstract

Classification problems having thousands or more classes naturally occur in NLP, for example language models or document classification. A softmax or one-vs-all classifier naturally handles many classes, but it is very slow at inference time, because every class score must be calculated to find the top class. We propose the "softmax tree", consisting of a binary tree having sparse hyperplanes at the decision nodes (which make hard, not soft, decisions) and small softmax classifiers at the leaves. This is much faster at inference because the input instance follows a single path to a leaf (whose length is logarithmic on the number of leaves) and the softmax classifier at each leaf operates on a small subset of the classes. Although learning accurate tree-based models has proven difficult in the past, we are able to overcome this by using a variation of a recent algorithm, tree alternating optimization (TAO). Compared to a softmax and other classifiers, the resulting softmax trees are both more accurate in prediction and faster in inference, as shown in NLP problems having from one thousand to one hundred thousand classes.

1 Introduction

Classification problems with thousands or more classes (sometimes called extreme classification) naturally occur in NLP and other areas. One example are language models. There are about 171k words in the current edition of the Oxford English Dictionary, and many more if we include all forms of a word, names, technical acronyms, etc. Another example is document classification. The Open Directory Project (ODP) contains over 1M website categories organized in a hierarchical ontology scheme. In this many-class setting, it is considerably difficult to learn a model that is accurate and fast at inference time. The simplest and most widespread model is a linear (e.g. softmax) classifier, possibly as the output layer of a neural net. One important problem with a softmax classifier is that one must compute the score or probability of (nearly) *all* classes, conditional on the input instance, in order to determine the (top-n) predicted class. This has a cost $\mathcal{O}(DK)$ where D is the input dimension of the softmax and K the number of classes, which is slow when K and D are large. This problem also occurs with other classifiers, such as soft decision trees. Indeed, computational constraints on the vocabulary size are a major challenge for neural machine translation (Koehn, 2020), for example.

We argue that having the classifier output a positive probability (however small) for each class is slow and unnecessary when K is large, because, for any given instance, the majority of classes should indeed have a negligible probability. A much faster classifier is a traditional decision tree, which makes hard decisions by thresholding an input feature at the decision nodes and outputs a single class at each leaf. This *axis-aligned tree* assigns zero probability to all classes except the predicted one, which is reached through a single root-leaf path very quickly (in $\log K$ time if the tree is balanced). However, such trees are known to be insufficiently accurate even if grown very deep.

We propose a *softmax tree (ST)*, a binary tree having sparse hyperplanes at the decision nodes (which make hard, not soft, decisions) and a small softmax classifier outputting $k < K$ classes at each leaf (a class may appear in more than one leaf). A ST is still very fast at inference: it sends the input instance to a single leaf via a path whose length is logarithmic on the number of leaves (for a complete tree), and it assigns (without computing them) probability zero to most classes (namely, all classes not in the leaf). Trading off the depth Δ of the tree and the number of classes k per leaf can potentially result in fast, highly accurate classifiers. However, STs are still hard to train because they define a nonconvex, nondifferentiable prob-

lem. We solve this by modifying *Tree Alternating Optimization (TAO)*, a recent algorithm for learning *oblique decision trees* (having hyperplane decision nodes and constant-label leaves), so that it can handle softmax leaves, and by using a good initialization.

Before describing the training algorithm (section 4), we review related work (section 2) and show (section 3) that oblique tree classifiers with constant-label leaves are intrinsically more powerful than linear classifiers, but possibly less efficient, which justifies our STs as hybrid tree-softmax classifiers. Finally, (section 5) we convincingly show that our STs have both higher accuracy and much faster inference time than several previous models on high-dimensional problems having up to 10^5 classes.

2 Related Work

The extreme multi-class classification problems have been previously addressed both in the literature of machine learning and natural language processing. The most basic method is one-versus-all (Bishop, 2006) where an independent binary classifier is learned per class. Another classical approach, error correcting output codes (ECOC) by Dietterich and Bakiri (1995), represents each class with a binary code and learns a separate binary classifier for each bit. However, these methods become costly or even intractable (both for training and predicting) when the number of classes is huge. Moreover, each binary classifier needs to handle highly imbalanced dataset as all classes except one would be instances of the negative class.

Various approaches have been proposed to speed up the training/prediction time and reduce the computational complexity. The most popular of them capitalize on using tree-based structures since it naturally leads to the logarithmic time reduction. Decision trees have been actively used in this area (Bengio et al., 2010; Daumé III et al., 2017). Nevertheless, traditional axis-aligned decision trees, such as C4.5 (Quinlan, 1993) or CART (Breiman et al., 1984), have very low accuracy (Choromanska and Langford, 2015). Nested dichotomies (Frank and Kramer, 2004) rely on a tree structure to divide a set of classes into two disjoint subsets and learn a binary classifier to separate them. However, human expertise is necessary to obtain a tree structure and class assignments. Additionally, the total error of the model accumulates over the depths since there is no way to refine binary classifiers once split is performed. More recent works which are specifically designed to cope with large number of classes (Beygelzimer et al., 2009; Bengio et al., 2010) employ similar idea but take into account class distributions to generate a tree structure. Other tree-based approaches include global or partial optimization over parameters of a tree. For instance, Daumé III et al. (2017) propose to use a fixed structured tree where each node has much smaller sized linear multi-class classifier. Sun et al. (2019) extend this work by allowing a tree structure to grow. Other works capitalize on generating "perfectly" balanced trees to guarantee logarithmic inference time (Jernite et al., 2017; Choromanska and Langford, 2015). Optimizing a tree parameters in these methods is typically done by approximating gradient information in a certain way (possibly in "online" fashion). Other tree-based methods exist with more focus on large scale extreme multi-label classification and ranking (Prabhu and Varma, 2014; Bhatia et al., 2015). Finally, there are works which employ non-tree based approaches, such as hashing-based methods (Medini et al., 2019), subsampling of classes and training set (Joshi et al., 2017), etc.

In the context of NLP, most of the above-mentioned methods are applicable in the number of practical applications, such as large-scale document classification and language modeling. Moreover, there are methods that are specifically designed for language modeling tasks where vocabulary size can be very large and it demands efficient computation of the softmax outputs. Hierarchical softmax (HSM) (Goodman, 2001) is an approximation which employs a "soft" decision tree with linear nodes to address this issue. HSM has been actively used in the problem of learning distributed representations of words (Bengio et al., 2003; Mikolov et al., 2013b) where it can be jointly trained with neural nets of various complexity (Morin and Bengio, 2005). Follow up works on this topic (Mnih and Hinton, 2009; Mikolov et al., 2013a) propose various initializations for the tree structure (e.g. random, Huffmann tree, etc.). Although the training of HSM can be efficiently done using specific loss functions, but during prediction time, input follows all children with a certain probability which brings no speedup compared to the plain softmax. It is still possible to transform a soft tree back into a "hard" tree

once training is done (by choosing a child with the highest probability at each split). For example, a recent work from Han et al. (2018) apply a similar approach. However, as we will experimentally show later, it increases an error due to further approximation. Similar observations were found in (Mikolov et al., 2013b) where various subsampling techniques showed better results. Recently, certain pruning mechanisms have been proposed as an alternative approach to speed up the prediction time (Bojanowski et al., 2017).

3 Linear Classifiers and Oblique Classification Trees

Define a K-class linear classifier $f(\mathbf{x}; \mathbf{A}, \mathbf{b})$: $\mathbb{R}^D \rightarrow \{1, \dots, K\}$ with parameters $\mathbf{A} \in \mathbb{R}^{K \times D}$ and $\mathbf{b} \in \mathbb{R}^K$ as $f(\mathbf{x}; \mathbf{A}, \mathbf{b}) = \arg\max(\mathbf{A}\mathbf{x} + \mathbf{b})$. The parameters are typically learned from data (e.g. by using the one-vs-all scheme or by optimizing a loss such as the cross-entropy for a softmax).

Theorem 3.1. *Any K-class linear classifier can be exactly represented by an oblique classification tree with constant-label leaves. The converse is not true.*

Proof. We give a constructive proof. Define $\mathbf{z} = \mathbf{A}\mathbf{x} + \mathbf{b} \in \mathbb{R}^K$, so the linear classifier output is $f(\mathbf{x}) = \arg\max(z_1, \dots, z_K)$. The latter argmax function of K arguments can be exactly computed by a complete binary tree of depth $K - 1$ as illustrated in fig. 1 for $K = 4$. Each decision node performs a comparison "$z_i \geq z_j$" and chooses the right child if it holds, else the left child. Each leaf's class label is the corresponding argmax value. This tree represents the iterative algorithm to compute $\arg\max(z_1, \dots, z_K)$ by scanning each element left to right for the maximum: $\max(z_K, \dots \max(z_3, \max(z_2, z_1)))$. Specifically, each root-leaf path tree corresponds to one possible execution path, and the nodes at depth j compare with z_j. Since a comparison "$z_i \geq z_j$" is equivalent to "$(\mathbf{a}_i - \mathbf{a}_j)^T\mathbf{x} + (b_i - b_j) \geq 0$", each decision node is a linear decision function and the tree is oblique. This argument is valid for interior points, but can be made to work for points on the boundary between classes by appropriately breaking the ties (replacing "$\geq$" with "$>$") as needed. The converse is not true because an oblique tree can dedicate more than one leaf to a class, resulting in a nonconvex class region (the union of two polytopes). This cannot be represented by a linear classifier, whose class regions are polytopes of the form $\mathbf{a}_i\mathbf{x} + b_i \geq \mathbf{a}_j\mathbf{x} + b_j \; \forall j \neq i$. □

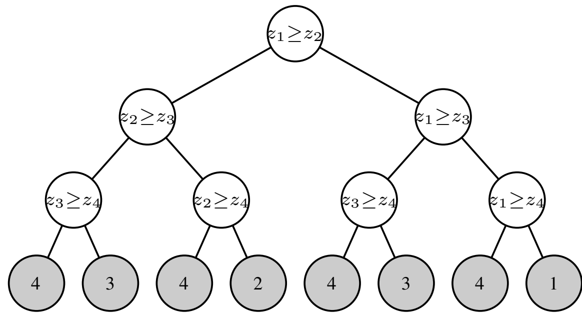

Figure 1: A linear classifier $f(\mathbf{x}) = \arg\max(\mathbf{A}\mathbf{x} + \mathbf{b}) \in \{1, \dots, K\}$ (with $K = 4$ classes in the diagram) can be exactly represented by an oblique decision tree with constant-label leaves, where $\mathbf{z} = \mathbf{A}\mathbf{x} + \mathbf{b}$.

Note that an axis-aligned decision tree (where each decision node tests a single input feature), however deep, cannot exactly represent a linear classifier unless the latter is itself axis-aligned.

The above theorem shows that oblique decision trees are strictly a more powerful family of classifiers than linear classifiers[1]. The proof construction produces a large tree, having 2^{K-1} leaves (although its inference time is equal to that of the linear classifier, $\mathcal{O}(DK)$). In practice, the tree need not exactly represent the linear classifier throughout the input space but just over the instances of a training set, and this can likely be achieved with far smaller trees. For example, if each class is linearly separable from the rest, the corresponding tree has just K leaves and each of the $K - 1$ decision nodes has exactly one leaf child (except the deepest decision node, which has two leaf children). Still, for a given dataset, the linear classifier may be a more efficient model in number of parameters than the tree. Which classifier (linear or oblique tree with constant-label leaves) is better is an empirical question. A further issue is that finding the global optimum of the training problem is easy for a linear classifier (e.g. the cross-entropy is convex for a softmax classifier) but NP-hard for a decision tree. This leads us to the model we propose in this paper, the softmax tree, which is a hybrid between a purely linear classifier and an oblique tree with constant-label leaves.

[1] It also follows from theorem 3.1 that an oblique decision tree with linear leaves is equivalent to a (somewhat) deeper oblique decision tree with constant leaves.

4 Softmax Trees: Definition and Training

Unlike soft decision trees, which can be readily optimized via gradient-based methods, hard decision trees pose a far more difficult optimization problem, not just nonconvex but nondifferentiable (and NP-hard). Traditional tree learning algorithms such as CART (Breiman et al., 1984) or C5.0 (Quinlan, 1993) are based on greedily and recursively partitioning the input space, and pruning the resulting tree to reduce overfitting. However, they are known to produce suboptimal trees (Hastie et al., 2009). In this work, we build on a recent algorithm, *Tree Alternating Optimization (TAO)* (Carreira-Perpiñán and Tavallali, 2018; Carreira-Perpiñán, 2021) which is a non-greedy optimization method for tree-based models. Originally described for oblique trees with constant-label leaves, TAO has shown a huge success in training a wide range of other tree-based models: regression trees (Zharmagambetov and Carreira-Perpiñán, 2020), tree ensembles (Carreira-Perpiñán and Zharmagambetov, 2020; Zharmagambetov et al., 2021a,b), hybrid models (Zharmagambetov and Carreira-Perpiñán, 2021a,b), etc. Moreover, they are shown to have a great potential to study model interpretability and explainability (Carreira-Perpiñán and Hada, 2021; Hada et al., 2021).

TAO works very differently from CART and much more like a regular machine-learning optimization algorithm, but instead of gradients (which do not apply) it uses alternating optimization over groups of nodes of a fixed tree structure. This results in a monotonic decrease of the objective function over all the tree parameters and convergence to a local optimum. Detailed comparison of TAO against traditional trees can be found in (Zharmagambetov et al., 2021c). Next, we describe our softmax trees and training algorithm, noting the differences with Carreira-Perpiñán and Tavallali (2018).

Consider a K-class problem with training set $\{(\mathbf{x}_n, y_n)\}_{n=1}^N \subset \mathbb{R}^D \times \{1,\dots,K\}$ of D-dimensional instances and labels. Let $\mathbf{T}(\mathbf{x};\mathbf{\Theta})$ be a binary decision tree which produces a prediction for each input $\mathbf{x}$ by routing $\mathbf{x}$ from the root to exactly one leaf and applying a predictor function at that leaf. Each node (both decision and a leaf) has learnable parameters $\boldsymbol{\theta}_i$ and the total set of parameters of a tree is $\mathbf{\Theta} = \{\boldsymbol{\theta}_i\}_{i\in\mathcal{N}}$, where $\mathcal{N}$ is the set of nodes. Each decision node i has a decision function $f_i(\mathbf{x};\boldsymbol{\theta}_i)$: $\mathbb{R}^D \rightarrow \{\texttt{left}_i, \texttt{right}_i\} \subset \mathcal{N}$, sending instance $\mathbf{x}$ to the corresponding child of node i, and each leaf has a predictor function $\mathbf{g}_i(\mathbf{x};\boldsymbol{\theta}_i)$: $\mathbb{R}^D \rightarrow \{1,\dots,K\}$ that produces the actual output. In a *softmax tree (ST)*:

- Each decision function uses a *(sparse) hyperplane* (*oblique tree*): "go to the right child if $\mathbf{w}_i^T\mathbf{x} + w_{i0} \geq 0$, else go to the left child", with parameters $\boldsymbol{\theta}_i = \{\mathbf{w}_i, w_{i0}\}$.
- Each leaf predictor is a k-class *linear softmax*: $\mathbf{g}_i(\mathbf{x};\boldsymbol{\theta}_i) = \sigma(\mathbf{W}_i\mathbf{x} + \mathbf{w}_i)$, where $\sigma(\cdot)$ is the softmax function and $\mathbf{W}_i \in \mathbb{R}^{k\times D}$, $\mathbf{w}_i \in \mathbb{R}^k$, where $k \leq K$ and usually $k \ll K$. This is unlike Carreira-Perpiñán and Tavallali (2018), which used a constant-label predictor.

TAO assumes a fixed tree structure (say, complete of depth Δ) and initial node parameters. Hence, the hyperparameters of a ST are Δ and k. TAO optimizes the following objective function:

$$E(\mathbf{\Theta}) = \sum_{n=1}^{N} L(\mathbf{y}_n, \mathbf{T}(\mathbf{x}_n;\mathbf{\Theta})) + \alpha \sum_{i\in\mathcal{N}} \|\boldsymbol{\theta}_i\|_1 \quad (1)$$

where $L(\cdot,\cdot)$ is the cross-entropy, and the ℓ_1 penalty over the weight vectors (of both decision nodes and leaves) promotes sparsity, via a hyperparameter $\alpha \geq 0$.

TAO is based on two theorems. First, eq. (1) *separates over any subset of non-descendant nodes* (e.g. all the nodes at the same depth); this follows from the fact that the tree makes hard decisions. All such nodes may be optimized in parallel. Second, optimizing over the parameters of a single node i simplifies to a well-defined *reduced problem* over the instances that currently reach node i (the *reduced set* $\mathcal{R}_i \subset \{1,\dots,N\}$). The form of the reduced problem depends on the type of node:

- For a decision node, it is a *weighted 0/1 loss binary classification problem*, where the two classes correspond to the left and right child, which are the only possible outcomes for an instance. Class $\texttt{left}_i$ ($\texttt{right}_i$) incurs a loss (weight) given by the prediction of the leaf reached from the left (right) child's subtree. Thus, each instance is assigned as *pseudolabel* the child with lower loss. The reduced problem takes the form (where $\overline{L}$ and $\overline{y}_n$ are the said loss and pseudolabel, resp.):

$$E_i(\boldsymbol{\theta}_i) = \sum_{n\in\mathcal{R}_i} \overline{L}(\overline{y}_n, f_i(\mathbf{x}_n;\boldsymbol{\theta}_i)) + \alpha \|\boldsymbol{\theta}_i\|_1. \quad (2)$$

This is as in Carreira-Perpiñán and Tavallali (2018) except that in our STs the loss is the cross-entropy of the corresponding leaf. This problem is NP-hard but can be well approximated with a convex surrogate; we use ℓ_1-regularized logistic regression where each instance is weighted by the loss difference between the winner child and the other child, and solve it using LIBLINEAR (Fan et al., 2008).

- For a leaf node, the reduced problem consists of optimizing the original loss but over the leaf classifier on its reduced set:

$$E_i(\boldsymbol{\theta}_i) = \sum_{n \in \mathcal{R}_i} L(\mathbf{y}_n, \mathbf{g}_i(\mathbf{x}_n; \boldsymbol{\theta}_i)) + \alpha \left\|\boldsymbol{\theta}_i\right\|_1. \quad (3)$$

In our STs, $\mathbf{g}_i$ is a k-class softmax classifier with an ℓ_1 sparsity penalty. We first estimate the k classes (out of K possible classes) as the k most populous classes in $\mathcal{R}_i$. Then we train the softmax, which is a convex problem. We solve it using SAG (Schmidt et al., 2017).

The resulting algorithm visits nodes in reverse breadt-first search order and is shown in Algorithm 1. Essentially, each iteration trains all nodes at the same depth (in parallel) from the leaves to the root, by solving either an ℓ_1-regularized softmax classifier at each leaf, or an ℓ_1-regularized logistic regression at each decision node. Note that the ℓ_1 penalty on the decision nodes' weight vectors means that some of them may become zero, which makes the node redundant and can be pruned at the end, reducing the size of the tree.

4.1 Dealing with zero probabilities

In our STs, each leaf operates on k classes. If $k = K$, each possible class receives a positive probability, but if $k \ll K$ then many $(K - k)$ classes receive exactly zero probability. This is necessary to achieve the fast prediction we seek, but it results in an infinite cross-entropy value whenever an instance with ground-truth class y is routed to a leaf that does not contain y. This causes no issue in the reduced problem over a leaf (the softmax uses only the top-k classes in that leaf), but it does cause an issue in the reduced problem over a decision node. Here, we have to solve a weighted 0/1 loss binary classification problem where the weights are obtained by evaluating the prediction's loss from the left and right subtrees for each instance in the node, and some of those weights can be infinity.

Algorithm 1: Softmax tree (ST) training.

```
Result: trained tree T(·; Θ)
input training set {(x_n, y_n)}_{n=1}^N;
  initial tree T(·; Θ) of depth Δ;
repeat
  for depth d = Δ downto 0 do
    for i ∈ nodes at depth d do
      if i is a leaf then
        R̄_i ← instances of the most
          populous k classes in R_i;
        θ_i ← fit a linear classifier on R̄_i
          to minimize eq. (3);
      else
        generate pseudolabels ȳ_n for each
          point n ∈ R_i;
        θ_i ← fit a weighted binary
          classifier to minimize eq. (2);
      end
    end
  end
until max number of iterations;
postprocessing: remove dead or pure subtrees;
```

To make sure learning succeeds, we tried the following approaches and evaluate their performances in Table 1:

1. Remove from the reduced problem any instance with loss=∞ (in either the left or right subtree). This performs very badly.

2. Replace loss=∞ by loss=β, where β is typically a large value (e.g. 100, 10^7). This is the option that works best in a number of datasets we have tried (see Table 1), but it requires an extra hyperparameter β. This is essentially the same as using a leaf model which predicts class probabilities with a softmax for its k classes and a constant, small value $\exp(-\beta)$ for all other $K - k$ classes.

3. Use the 0/1 loss instead of the cross-entropy in the overall objective function of eq. (1). This avoids the infinity issue altogether, since the pseudolabels' weight is either 0 or 1 (as in Carreira-Perpiñán and Tavallali, 2018). However, the reduced problem over a leaf must now optimize the 0/1 loss (which is NP-hard) rather than the cross-entropy; we approximate this by using the cross-entropy as surrogate loss, so we still learn a softmax as usual. This requires no additional hyperparameter and does quite well. It is our default option (unless otherwise specified in sec. 5).

	Method	top-1_{train} (%)	top-1_{test} (%)
ALOI	remove loss=∞	97.37	97.80
	0/1 loss	4.70	12.51
	∞–to–10^7	4.27	11.49
	∞–to–100	4.15	11.22
WIKI-Small	remove loss=∞	95.51	96.05
	0/1 loss	66.95	76.33
	∞–to–10^7	64.47	76.24
	∞–to–100	64.47	76.07

Table 1: Top-1 errors for STs with different ways of handling the loss=∞ during the decision node optimization. "0/1 loss" refers to using the 0/1 loss instead of the cross-entropy, "remove loss=∞" refers to removing the instances with loss= ∞, and "∞-to-β" refers to approximating the infinity loss as β (e.g. $\beta = 100$).

4.2 Obtaining an initial tree

While TAO monotonically decreases the objective function, it still converges to a local optimum. For the constant-label leaf oblique trees of Carreira-Perpiñán and Tavallali (2018), which were applied to problems with few classes, using as initial tree a complete tree of depth Δ with random parameters worked well (we call this "random initialization"). However, with many classes we have observed that the following *greedy hierarchical clustering* initialization works quite better. Assume a complete tree of depth Δ having $L = 2^{\Delta}$ leaves (although the idea carries over to any binary tree structure). The following simple algorithm is guaranteed to assign classes to leaves in a way that respects the ST structure and keeps similar classes near each other in the tree (pseudocode can be found in Appendix A).

First, we cluster the training instances into L clusters using k-means. The L clusters will be assigned one-to-one to the L leaves by a greedy hierarchical clustering, as follows. We greedily merge pairs of clusters to achieve $\frac{L}{2}$ "superclusters". That is, we first merge the two closest clusters into one supercluster (which becomes their parent node). Then, we merge the two closest clusters of the remaining clusters, etc. Note that, unlike in regular hierarchical agglomerative clustering, the resulting supercluster is not considered for merging immediately, but rather each level is considered separately, so that we obtain a tree with a desired structure (balanced). We define the distance between two (super)clusters as the Euclidean distance between their means. We repeat the greedy merging into $\frac{L}{4}$, $\frac{L}{8}$, etc. superclusters until we reach a single supercluster containing all training instances (the root of the tree). This gives the assignment of clusters to leaves of our tree. (A faster version of this is obtained by first replacing all the training instances within each class with a "class prototype", weighted by the number of instances, and then proceeding as above to find a greedy hierarchical clustering of these K prototypes.) Now that each instance is assigned to one leaf, the first TAO iteration can start, in reverse BFS order.

The idea is that the tree leaves induce a hierarchical partition of the input space into polytopes, hence 1) the training instances within one leaf's polytope should generally be closer to each other than to instances in other polytopes, and 2) this remains true as clusters are merged according to the tree (i.e., the polytopes of two sibling leaves will be near each other, etc.).

4.3 Computational complexity

Training Assuming training each node (logistic regression or softmax) is linear on the sample size, training all the decision nodes at the same depth is approximately constant and equal to training one logistic regression on the whole training set; likewise, training all the leaves is approximately equal to training one k-class softmax classifier on the whole training set. Thus, the total *sequential* cost of one iteration is approximately equal to that of one k-class softmax and Δ logistic regressions on the whole dataset. As noted above, all the nodes at the same depth can be trained *in parallel*.

Inference Assuming the final tree is complete, an input instance spends $\mathcal{O}(\Delta D)$ to reach a leaf and $\mathcal{O}(kD)$ at its softmax (which typically dominates the path cost). This is a speedup of $\mathcal{O}(\frac{K}{\Delta+k}) \approx \mathcal{O}(\frac{K}{k})$ compared to a single softmax over all classes, a remarkable speedup in practice. The inference time is actually smaller because 1) the final tree may be smaller because some nodes were pruned, and 2) the weight vectors at decision node hyperplanes and leaf softmaxes are typically sparse (this is particularly important with high-dimensional features such as bag-of-words).

5 Experiments

We demonstrate the performance of our method on two popular NLP tasks: (a) large scale text classification, and (b) language modeling. Experiments suggest that *our resulting softmax trees outperform simple and advanced baselines either in accuracy (and yet very fast) or in prediction time*

(and yet showing competitive accuracy); or quite often in both of these indicators. Moreover, the resulting models are compact in terms of memory requirements.

5.1 Setup

We initialize our softmax trees (ST) using a "clustering-based" method described in section 4.2 (unless otherwise specified). The sparsity penalty (α) set according to the cross-validation (10% of the training data). Increasing the number of TAO iterations results to a better performance but at a cost of having slower training time. Maximum number of classes (k) at each leaf is another tunable hyperparameter and we report it for each performed experiment (e.g. ST(k=50)). Details and exact values for all other hyperparameters can be found in Appendix C.

As for the baselines, we use scikit-learn's (Pedregosa et al., 2011) implementation of the one-versus-all and softmax linear classifiers. Additionally, we compare our results with more recent baselines which show state-of-the-art performance on various extreme classification problems: LOMTree (Choromanska and Langford, 2015), RecallTree (Daumé III et al., 2017), (π, κ)-DS (Joshi et al., 2017) and MACH (Medini et al., 2019). Where applicable, we use the available implementations of the mentioned methods. Finally, we have implemented hierarchical softmax as a tree-based baseline for language modeling tasks. Further details can be found in Appendix C.

We report the top-1 and top-5 errors, maximum depth (Δ), mean inference time per test sample (in ms) and uncompressed model sizes (in GB). We average the errors over 3 independent runs for softmax trees, whereas the best performance is reported for other baselines. The inference time is calculated in a single CPU without parallel processing using the following methodology: we sequentially pass each test sample to the trained model and measure its prediction time. Then we average the results over all test set. Also, we report the storage requirement for each model (uncompressed and stored in sparse format if applicable). Appendix D has additional metrics (e.g. number of leaves, number of classes per leaf, etc.). Our hardware setup is Intel Xeon CPU E5-2699 v3 @ 2.30GHz with 256 GB RAM.

	Method	top-1	Δ	inf.(ms)	size(GB)
WIKI–Small	RecallTree	92.64	15	0.97	0.8
	one-vs-all	85.71	0	10.70	53.5
	MACH	84.80	–	252.64	1.3
	(π, κ)-DS	78.02	–	10.33	0.01
	ST($k = 100$)	77.26	7	0.33	0.03
	ST($k = 300$)	76.86	7	0.49	0.04
	ST($k = 150$)	76.33	8	0.57	0.05
	ST$^+$($k = 150$)	75.65	8	0.52	0.05
ODP	RecallTree	94.64	6	8.42	3.4
	LOMTree	(93.46)	(17)	(0.26)	–
	one-vs-all	89.22	0	1317.58	155.7
	(π, κ)-DS	86.31	–	36.41	1.0
	MACH	84.55	–	684.04	1.2
	ST($k = 300$)	83.78	9	9.59	0.1
	ST$^+$($k = 300$)	81.84	9	9.87	0.1

Table 2: Results on text classification datasets. We report the top-1 test error (see App. D for top-5), maximum depth (Δ), avg. inference time per test sample (in ms) and uncompressed model sizes (in GB). ST($k = x$) indicates our method which uses at most k classes at each leaf. The results in brackets are taken from the corresponding papers. "+" shows the results of using cross-entropy loss with $\beta = 100$ (see section 4.1).

5.2 Results: text classification

We perform the first set of experiments on two document categorization benchmarks with large number of classes: ODP–website categorization problem which has over 105k classes and WIKI–Small (with $>$ 36k classes). Input feature vector for each document is normalized bag-of-words representation containing around 400k dimensions. See Appendix B for details and additional benchmarks.

Table 2 shows that the STs consistently outperform other baselines and by a considerable margin, showing outstanding performance on these benchmarks. Moreover, they achieve faster inference time compared to most of the baselines (e.g. one-vs-all, MACH) and shows a similar speed as of RecallTree and LOMTree (i.e., other tree-based methods). It worth to mention that our obtained inference times for some baselines (e.g. RecallTree, MACH) diverge from the reported results in other papers. We believe this is because: 1) different computing setup is used; 2) measuring methodology is somewhat different (see setup).

Additionally, fig. 2 shows a tradeoff between error-vs-depth and inference time-vs-depth. It also examines different values for k. In general, increasing k results to better models in terms of error. On the other hand, it increases the inference time (right figure), although the difference is typically negligible. Finally, the results suggest that

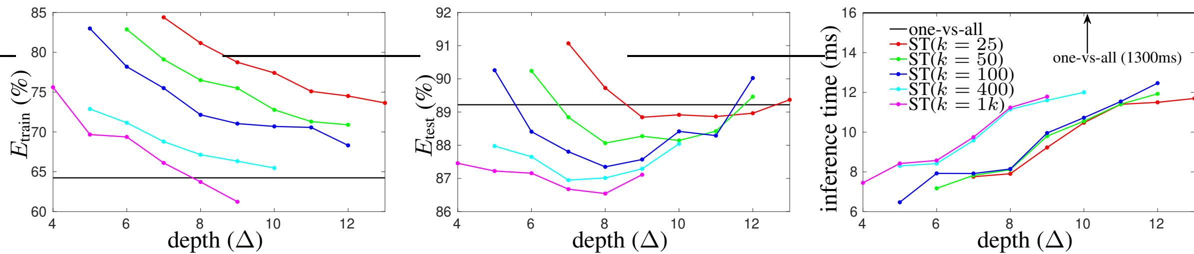

Figure 2: Top-1 errors and avg. inf. time tradeoff of the ST for various settings of Δ and k on the ODP dataset.

Method	top-1/top-5	PPL(% covered)	Δ	inf.(ms)
HSM-approx	92.2 / 86.5	575 (100%)	18	0.184
HSM	91.1 / 81.1	575 (100%)	18	0.421
one-vs-all	87.5 / 80.2	220 (100%)	0	0.402
softmax	86.9 / 79.6	217 (100%)	0	0.467
ST(k=50)	86.5 / 72.5	17 (44%)	8	0.058
ST(k=100)	86.5 / 71.5	27 (51%)	7	0.058
ST(k=200)	86.4 / 70.6	45 (58%)	6	0.053
ST(k=400)	86.4 / 69.7	71 (67%)	5	0.064
ST(k=800)	86.4 / 68.4	117 (77%)	4	0.066
ST*(k=800)	86.4 / 68.4	427 (100%)	4	0.066

Table 3: Like Table 2 but on PTB–language modeling task. We also report the test Perplexity (with percentage of the covered points) and top-5 error. "*" indicates that smoothing was applied to replace 0 probabilities with some small epsilon and renormalize the output.

Method	top-1/top-5	PPL(% covered)	Δ	inf.(ms)
HSM-appox	78.3 / 64.1	184 (100%)	18	0.097
HSM	77.7 / 63.1	184 (100%)	18	0.372
softmax	74.3 / 54.8	96 (100%)	0	0.346
ST(k=50)	75.2 / 57.3	9 (59%)	8	0.046
ST(k=100)	75.0 / 56.8	13 (64%)	7	0.045
ST(k=200)	74.9 / 56.2	18 (70%)	6	0.067
ST(k=400)	74.7 / 55.9	24 (76%)	5	0.066
ST(k=800)	74.5 / 55.5	33 (81%)	4	0.069
ST*(k=800)	74.5 / 55.5	145 (100%)	4	0.069

Table 4: Like Table 3, but models were trained on the output of the recurrent neural net (LSTM).

the Depth (Δ) should be sufficiently large but overfitting may occur passing a certain point (e.g. middle plot). Note that for these set of experiments, we use a random initialization for STs.

Model sizes Table 2 reports another critical aspect – compactness of our models. Just as our STs are very fast, they also generate extremely compact models compared to baselines (at least 10x gain). This is due to the L1 penalty applied at each node, which leads to sparse weights. Moreover, we observe that the best performance for STs is typically achieved with shallow trees (see Δ) which also helps to reduce the model size.

5.3 Results: language modeling

We conduct experiments on PTB dataset which has been extensively used to study language modeling problems. Dataset description as well as our preprocessing steps can found in Appendix B. As for the baselines, we use the same one-vs-all classifier described earlier and Hierarchical Softmax (HSM) model (we closely follow the setup from (Mikolov et al., 2013a)). Also, we have implemented "HSM-approx" which chooses a child with the highest probability at each split (i.e., it achieves a faster prediction time). Setup for one-vs-all and ST is the same as in section 5.1, except we use the random initialization for ST. As for the HSM, we use our own implementation in Pytorch (see details in Appendix C).

We report the train/test Perplexities (PPL), which is commonly done for such tasks: PPL $= \exp(-\frac{1}{N}\sum_{i=1}^{N}\log Pr(y_i|\mathbf{x}_i))$, where N is the sample size (train or test), y_i and $\mathbf{x}_i$ are ground truth label and input feature vector of the instance i, respectively. Most of the baselines described in the previous section (especially tree-based methods) do not produce class probabilities and they can not be directly applied to solve the language modeling problem, so we omit their comparison. For ST, we calculate $Pr(y_i|\mathbf{x}_i)$ by routing an instance x_i to the corresponding leaf of a tree and taking softmax on the output produced by that leaf. If y_i (correct class) is not presented in that leaf (it may happen since a leaf stores $k < K$ classes) then we do not include it to the calculation. Therefore, we provide the total number of points with non-zero probability predictions. Note that the fact that our STs output exactly zero probability for many classes is by design and results in its inference speed. Also note that a softmax classifier will happily assign a positive probability to a class whose region is actually empty (i.e., no input $\mathbf{x} \in \mathbb{R}^D$ ever results in that class winning). That said, we also provide the results of applying

a smoothing technique (Eisenstein, 2019, section 6.2) to ensure positive probabilities for all classes, without any increase in inference time (denoted by "*" in tables and more results can be found in Appendix D). Specifically, we assign some small ϵ to all instances with zero probability and renormalize the output probabilities. This requires additional hyperparameter ϵ which we tune using cross-validation.

Table 3 summarizes our results. First of all, one can notice that both versions of HSM perform worse compared to one-vs-all (both error and PPL) which coincides with previous findings (Mikolov et al., 2013b). As for the ST, it shows a decent test error (both top-1/top-5) and the fastest inference time than the other baselines. Regarding the perplexity score, our method produces exactly zero probability for some instances which makes overall PPL unbounded (i.e., infinity). However, if we discard such cases and focus on a subset of data for which probability estimate is non-zero (see "% covered" in the table), then it achieves a significantly low PPL. Moreover, it is clear from the Table 3 that ST covers majority of the points and such coverage increases as we increase k. As for the results using smoothing (denoted by "*"), the PPL score is still much lower compared to HSM but higher than one-vs-all. This logically makes sense since instances with zero probability increase PPL score substantially.

5.3.1 Neural language modeling

Modern neural nets are well known to achieve the state-of-the-art performance in language modeling problems. As a comparison, simple RNNs can easily reach PPL $=$ 101 on the same problem (Mikolov et al., 2011) from the previous section. Therefore, we combine our softmax trees with the output of LSTM and show that it achieves a comparable performance with faster inference time. Specifically, we use our Pytorch implementation (see details in Appendix C) of the RNN model for the word-level language modeling on the same PTB dataset with all 10k unique words as the vocabulary. Table 4 summarizes our findings. The neural net model achieves 96.33 perplexity score on a test set using softmax classifier as the last layer. Once training is done, we extract the last output of the LSTM layer and use it as input to the ST (i.e., input is a vector $\in \mathbb{R}^{150}$). In other words, ST is not trained in end-to-end fashion but sequentially. Despite this, our method shows a similar performance compared to the plain softmax in terms of train/test errors and consistently faster during inference time (about 5.7 times). Regarding the perplexity score, as in the above case, we cover majority of the data points for which the PPL is significantly low compared to the baseline.

Method	WIKI–Small	ODP
one-vs-all	>7d	>7d
LOMTree	–	(36m)
RecallTree	53m	113m
MACH	1445m	2301m
ST	1033m	2880m

Table 5: Training times in minutes (m) or days (d) for the datasets in Table 2. For ST, we report the training times for the best performing architecture (in terms of test error). For LOMTree, we report the results from (Daumé III et al., 2017) when applicable.

5.4 Training time

Table 5 gives representative runtimes for several datasets. We train all methods using at most 16 parallel threads. In general, all tree-based and hashing-based methods are faster to train compared to one-vs-all. For smaller datasets (see Appendix D), training softmax trees as expensive as RecallTree, but faster than MACH. For larger datasets, ST requires more time to find a good solution. Even in that case, it shows a comparable runtime against MACH. Overall, the runtime of ST is reasonable and more than justified by the fast inference time and low test error it achieves.

6 Conclusion

Softmax trees strike a balance between having a single softmax classifier, which is easy to optimize but slow at inference, and a decision tree with contant-label leaves, which is hard to optimize but fast at inference. Tuning the depth of the tree and the number of classes per leaf softmax results in classifiers that are both more accurate and much faster than a regular softmax or other hierarchical softmax approaches in many-class problems. Finding good local optima for softmax trees is possible with a modification of the tree alternating optimization (TAO) algorithm combined with a good initialization. We are now working on forests of softmax trees and on growing the tree structure adaptively.

Acknowledgments. Work supported by NSF award IIS–2007147.

Broader Impact

Our work is on optimization and efficiently training of machine learning models for classification task. We anticipate no impact beyond that of the models themselves.

References

Samy Bengio, Jason Weston, and David Grangier. 2010. Label embedding trees for large multi-class tasks. In *Advances in Neural Information Processing Systems (NIPS)*, volume 23, pages 163–171. MIT Press, Cambridge, MA.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *J. Machine Learning Research*, 3(1137–1155).

Alina Beygelzimer, John Langford, and Pradeep Ravikumar. 2009. Error-correcting tournaments. In *International Conference on Algorithmic Learning Theory (ALT 2009)*, pages 247–262.

Kush Bhatia, Himanshu Jain, Purushottam Kar, Manik Varma, and Prateek Jain. 2015. Sparse local embeddings for extreme multi-label classification. In *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 730–738. MIT Press, Cambridge, MA.

Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer Series in Information Science and Statistics. Springer-Verlag, Berlin.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Trans. Association for Computational Linguistics*, 5:135–146.

Leo J. Breiman, Jerome H. Friedman, R. A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees*. Wadsworth, Belmont, Calif.

Miguel Á. Carreira-Perpiñán. 2021. The Tree Alternating Optimization (TAO) algorithm: A new way to learn decision trees and tree-based models. ArXiv.

Miguel Á. Carreira-Perpiñán and Suryabhan Singh Hada. 2021. Counterfactual explanations for oblique decision trees: Exact, efficient algorithms. In *Proc. of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, pages 6903–6911, Online.

Miguel Á. Carreira-Perpiñán and Pooya Tavallali. 2018. Alternating optimization of decision trees, with application to learning sparse oblique trees. In *Advances in Neural Information Processing Systems (NEURIPS)*, volume 31, pages 1211–1221. MIT Press, Cambridge, MA.

Miguel Á. Carreira-Perpiñán and Arman Zharmagambetov. 2020. Ensembles of bagged TAO trees consistently improve over random forests, AdaBoost and gradient boosting. In *Proc. of the 2020 ACM-IMS Foundations of Data Science Conference (FODS 2020)*, pages 35–46, Seattle, WA.

Anna E Choromanska and John Langford. 2015. Logarithmic time online multiclass prediction. In *Advances in Neural Information Processing Systems (NIPS)*, volume 28. MIT Press, Cambridge, MA.

Hal Daumé III, Nikos Karampatziakis, John Langford, and Paul Mineiro. 2017. Logarithmic time one-against-some. In *Proc. of the 34th Int. Conf. Machine Learning (ICML 2017)*, pages 923–932, Sydney, Australia.

Aaron Defazio, Francis R. Bach, and Simon Lacoste-Julien. 2014. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems (NIPS)*, volume 27, pages 1646–1654. MIT Press, Cambridge, MA.

Thomas G. Dietterich and G. Bakiri. 1995. Solving multi-class learning problems via error-correcting output codes. *J. Artificial Intelligence Research*, 2:253–286.

Jacob Eisenstein. 2019. *Introduction to Natural Language Processing*. MIT Press.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *J. Machine Learning Research*, 9:1871–1874.

Eibe Frank and Stefan Kramer. 2004. Ensembles of nested dichotomies for multi-class problems. In *Proc. of the 21st Int. Conf. Machine Learning (ICML'04)*, pages 305–312, Banff, Canada.

Joshua Goodman. 2001. Classes for fast maximum entropy training. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'01)*, pages 561–564, Salt Lake City, Utah, USA.

Suryabhan Singh Hada, Miguel Á. Carreira-Perpiñán, and Arman Zharmagambetov. 2021. Sparse oblique decision trees: A tool to understand and manipulate neural net features. ArXiv:2104.02922.

Lei Han, Yiheng Huang, and Tong Zhang. 2018. Candidates vs. noises estimation for large multi-class classification problem. In *Proc. of the 35th Int. Conf. Machine Learning (ICML 2018)*, pages 1890–1899, Stockholm, Sweden.

Trevor J. Hastie, Robert J. Tibshirani, and Jerome H. Friedman. 2009. *The Elements of Statistical Learning—Data Mining, Inference and Prediction*, second edition. Springer Series in Statistics. Springer-Verlag.

Yacine Jernite, Anna Choromanska, and David Sontag. 2017. Simultaneous learning of trees and representations for extreme classification and density estimation. In *Proc. of the 34th Int. Conf. Machine Learning (ICML 2017)*, pages 1665–1674, Sydney, Australia.

Bikash Joshi, Massih R. Amini, Ioannis Partalas, Franck Iutzeler, and Yury Maximov. 2017. Aggressive sampling for multi-class to binary reduction with applications to text classification. In *Advances in Neural Information Processing Systems (NIPS)*, volume 30. MIT Press, Cambridge, MA.

Philipp Koehn. 2020. *Neural Machine Translation*. Cambridge University Press.

Tharun Kumar Reddy Medini, Qixuan Huang, Yiqiu Wang, Vijai Mohan, and Anshumali Shrivastava. 2019. Extreme classification in log memory using count-min sketch: A case study of Amazon search with 50M products. In *Advances in Neural Information Processing Systems (NEURIPS)*, volume 32, pages 13265–13275. MIT Press, Cambridge, MA.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient estimation of word representations in vector space. In *ICLR Workshop*.

Tomas Mikolov, Anoop Deoras, Stefan Kombrink, Lukáš Burget, and Jan Černocký. 2011. Empirical evaluation and combination of advanced language modeling techniques. In *Proc. of Interspeech'11*, pages 605–608, Florence, Italy.

Tomas Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proc. of Interspeech'10*, pages 1045–1048, Makuhari, Japan.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NIPS)*, volume 26, pages 3111–3119. MIT Press, Cambridge, MA.

Andriy Mnih and Geoffrey E. Hinton. 2009. A scalable hierarchical distributed language model. In *Advances in Neural Information Processing Systems (NIPS)*, volume 21, pages 1081–1088. MIT Press, Cambridge, MA.

Frederic Morin and Yoshua Bengio. 2005. Hierarchical probabilistic neural network language model. In *Proc. of the 10th Int. Workshop on Artificial Intelligence and Statistics (AISTATS 2005)*, pages 246–252, Barbados.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging AI applications. In *Proc. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

Ioannis Partalas, Aris Kosmopoulos, Nicolas Baskiotis, Thierry Artières, George Paliouras, Éric Gaussier, Ion Androutsopoulos, Massih-Reza Amini, and Patrick Gallinari. 2015. LSHTC: A benchmark for large-scale text classification. ArXiv:1503.08581.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine learning in Python. *J. Machine Learning Research*, 12:2825–2830. Available online at https://scikit-learn.org.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *Proc. ACL–14 Conf. Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 1532–1543, Doha, Qatar.

Yashoteja Prabhu and Manik Varma. 2014. FastXML: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *Proc. of the 20th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (SIGKDD 2014)*, pages 263–272, New York, NY.

J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

M. Schmidt, Nicolas Le Roux, and Francis R. Bach. 2017. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162:83–112.

Wen Sun, Alina Beygelzimer, Hal Daum'e III, John Langford, and Paul Mineiro. 2019. Contextual memory trees. In *Proc. of the 36th Int. Conf. Machine Learning (ICML 2019)*, pages 6026–6035, Long Beach, CA.

Arman Zharmagambetov and Miguel Á. Carreira-Perpiñán. 2020. Smaller, more accurate regression forests using tree alternating optimization. In *Proc. of the 37th Int. Conf. Machine Learning (ICML 2020)*, pages 11398–11408, Online.

Arman Zharmagambetov and Miguel Á. Carreira-Perpiñán. 2021a. Learning a tree of neural nets. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'21)*, pages 3140–3144, Toronto, Canada.

Arman Zharmagambetov and Miguel Á. Carreira-Perpiñán. 2021b. A simple, effective way to improve neural net classification: Ensembling unit activations with a sparse oblique decision tree. In *IEEE Int. Conf. Image Processing (ICIP 2021)*, pages 369–373, Online.

Arman Zharmagambetov, Magzhan Gabidolla, and Miguel Á. Carreira-Perpiñán. 2021a. Improved boosted regression forests through non-greedy tree

optimization. In *Int. J. Conf. Neural Networks (IJCNN'21)*, Virtual event.

Arman Zharmagambetov, Magzhan Gabidolla, and Miguel Á. Carreira-Perpiñán. 2021b. Improved multiclass AdaBoost for image classification: The role of tree optimization. In *IEEE Int. Conf. Image Processing (ICIP 2021)*, pages 424–428, Online.

Arman Zharmagambetov, Suryabhan Singh Hada, Magzhan Gabidolla, and Miguel Á. Carreira-Perpiñán. 2021c. Non-greedy algorithms for decision tree optimization: An experimental comparison. In *Int. J. Conf. Neural Networks (IJCNN'21)*, Virtual event.

A Pseudocode for Obtaining an Initial Tree Structure

Algorithm 2: Generating an initial tree

```
Result: inital reduced sets R
input training set {(x_n, y_n)}_{n=1}^N, depth Δ ;
for i ∈ {1, ..., K} do
  | μ_i ← average feature vector for class i
end
C^Δ := {C_i^Δ}_{i=1}^{2^Δ} ← K-means clusters^2 on {μ_i}_{i=1}^K ;
for depth d = Δ up to 1 do
  for i ∈ {1, ..., 2^{d-1}} do
    Find and remove the closest two clusters
      (C_m^d, C_n^d) ∈ C^d based on the mean ;
    Merge (C_m^d, C_n^d) to a parent cluster C_i^{d-1} ;
  end
end
Map clusters C to the corresponding reduced sets R
```

B Datasets

Dataset	N_{train}	N_{test}	D	K
ALOI	97 200	10 800	128	1 000
PTB	400 097	34 633	150	5 970
WIKI–Small	796 617	199 155	380 078	36 504
ODP	1 084 404	493 014	422 712	105 033

Table 6: Datasets used in our experiments: number of points for training and test (N_{train}, N_{test}), number of features D and number of classes K.

Table 6 summarizes the characteristics of the datasets used in our experiments. Below we provide a description for each of them.

- **ALOI** (Amsterdam Library of Object Images) is a color image collection of one-thousand small objects, recorded for scientific purposes. Images for each object category are created by systematically changing viewing angle, illumination angle, and illumination color (see details at https://aloi.science.uva.nl/). We obtained the preprocessed form of the dataset from the LIBSVM multiclass data collection, where the extended color histogram with 128 dimensions is used to extract image features. We follow the same random partition of the data (90% train and 10% test) as in (Choromanska and Langford, 2015). As a preprocessing step, we subtract the mean.

- **ODP** (Open Directory Project) is the comprehensive human-edited directory of the website categories. As of April 2013 there were over 1M categories organized in a hierarchical ontology scheme. We use the preprocessed version of it[3] which uses 105k categories, as in (Daumé III et al., 2017; Medini et al., 2019). For each document, input feature vector is bag-of-words (normalized) and the class label is the category associated with the document.

- **WIKI–Small** is another text categorization dataset obtained from Joshi et al. (2017). It is a subset of Large Scale Hierarchical Text Classification challenge (LSHTC) (Partalas et al., 2015). For each document, a feature vector is bag-of-words and the class label is the category associated with the document obtained from DMOZ and DBpedia hierarchical ontology of the WEB.

- **PTB** (Penn Treebank) is a standard dataset used to evaluate performances of language models. We use the preprocessed version from Mikolov et al. (2010) which is publicly available online. The dataset consists of the plain text sentences in English with approximately 1M tokens and 10k unique words (i.e. vocabulary size). For the neural language modeling experiments, we proceed with this dataset as is without further modification (i.e. section 5.3.1). But for the section 5.3, we construct the dataset as follows. We filter out words that appeared less than 10 times which leaves us with 5 970 unique words (=number of classes). We construct a multiclass classification task as predicting the next word given previous 3 words. As for the input features, we use a pretrained version of GloVe (Pennington et al., 2014)[4] to obtain a word representation in vector space. We downloaded pretrained word vectors ($\in \mathbb{R}^{50}$) which were trained on Wikipedia 2014 and Gigaword 5. We obtain a word vector for each context word and simply concatenate them. For example, consider the following sequence "black lives matter protest". First, we extract 50 dimensional GloVe vectors for "black", "lives", "matter" and then concatenate them which results into 150 dimensional vector (i.e. this would be the total number of input features). The ground truth label would be a single integer: 4011 (assuming the index of the word "protest" is 4011 in our vocabulary).

[2] Alternatively, to obtain the leaf clusters, we can run $\mathcal{K}$-means on the whole dataset.

[3] http://hunch.net/~vw/odp_train.vw.gz, http://hunch.net/~vw/odp_test.vw.gz

[4] https://nlp.stanford.edu/software/

C Hyperparameter Tuning for ST and Baselines

- **ST** We have implemented our softmax trees in Python 3.8.3 with parallel processing at each level (using Ray (Moritz et al., 2018)). The sparsity penalty (α) set according to the cross-validation (usually 10% of the training data). Experimentally, we have found that $\alpha = 0.1$ (for ODP, ALOI) and $\alpha = 1.0$ (for PTB, WIKI–Small) leads to the best performance. We report the mean error (training and test) and standard deviation over 3 independent runs (in most cases) or result of a single run. A number of TAO iterations is set to 20 for PTB and ODP; 30 for ALOI; and 40 for WIKI–Small. A decision node optimization involves an ℓ_1-regularized logistic regression which is solved using LIBLINEAR (Fan et al., 2008). Similarly, optimizing a single leaf involves ℓ_1-regularized k-class linear classification which is solved using SAGA (Defazio et al., 2014). Both SAGA and LIBLINEAR are available through scikit-learn interface (Pedregosa et al., 2011). For SAGA, we set the maximum number of iterations to 20. For the $\mathcal{K}$-means clustering algorithm, we use a Python implementation available in scikit-learn. We use default parameters, except for the number different runs `n_init` for the ODP dataset to make the runtime faster. Finally, we experiment with several types of losses as explained in section 4.1. In most of our experiments, we use 0/1 loss to approximate loss=∞ and this is our default option. However, we found that carefully tuned β (e.g. 100) for cross-entropy loss shows the best results for text classification tasks and we use it to report our final results.

- **One-vs-all and linear softmax** We use scikit-learn's implementation for these baselines (with l2 penalty and "SAG" (Schmidt et al., 2017) solver since it is scalable to larger datasets). We set `n_jobs` parameter to 32 and `C` to 10 in all of our experiments. Since running one-versus-all takes extremely large runtime, we limit the maximum number of iterations (10 for ODP, 100 for ALOI, 20 for WIKI–Small and PTB). We do not report results of the linear softmax for text classification since: 1) it shows similar performance as one-vs-all; 2) and it requires a huge resources to run for ODP and Wiki-Small (simply infeasible for our hardware setup).

- **MACH** (Medini et al., 2019) We use their available implementation online[5] and tune its most important hyperparameters (B, R) for each dataset. See Table 2 for the specific values for each problem.

- **(π, κ)-DS** (Joshi et al., 2017) We use their available implementation online with the set of hyperparameters suggested by authors.

- **RecallTree** (Daumé III et al., 2017) We use a version implemented inside Vowpal Wabbit[6]. For ODP and ALOI, we use the suggested hyperparameters from the official web page. However, we tune its most important hyperparameters for WIKI–Small: `max_candidates`, `max_depth`, `passes`.

- **Hierarchical Softmax (HSM)** We use our own implementation in Pytorch. We closely follow the setup from (Mikolov et al., 2013a): the structure of a tree is obtained from Huffman's algorithm (frequency of each word is calculated from the raw training data), each decision node applies a linear transformation followed by sigmoid non-linearity and the objective function to minimize is negative log-likelihood. Training is done using SGD with small learning rate of 0.005 multiplied by 0.995 after each step and fixed momentum of 0.9. HSM, in its pure implementation, does not support mini-batch updates (i.e. > 1, although various methods exist to agglomerate gradients for each node) and thus, we set it to 1.

- *Training LSTM for language modeling*. Our implementation is similar to the one that can be found in the official Pytorch examples web page. We choose LSTM model with two layers, with embedding size of 256 and 150 hidden states. The sequence length is fixed as 20 and an initial learning rate for SGD is set to 20 which is divided by 4 if no improvement on the validation loss. We train this model for 40 epochs using negative log-likelihood as a criterion.

D Extended Results and Additional Experiments

This section presents extended tables/figures and results on additional datasets. For all experiments

[5] https://github.com/RUSH-LAB/MACH
[6] https://vowpalwabbit.org/

	Method	top-1 (%)	top-5 (%)
ALOI	ST($k = 30, \Delta = 7$)	10.47	4.81
	ST($k = 1000, \Delta = 3$)	8.45	2.09
Wiki Small	ST($k = 100, \Delta = 8$)	77.26	59.89
	ST($k = 150, \Delta = 8$)	77.07	57.91
ODP	ST($k = 300, \Delta = 9$)	83.78	76.04

Table 7: Top-1 (%) and top-5 (%) errors for selected models on text classification benchmarks.

Method	ALOI	WIKI–Small	ODP
one-vs-all	111m	>7d	>7d
LOMTree	2m	–	36m
RecallTree	83m	53m	113m
MACH	43m	1445m	2301m
ST	37m	495m	3072m
ST[†]	127m	1033m	2880m

Table 8: Training times: similar to Table 5 but extended with the results on ALOI.

below, we denote by "ST[†]" our special initialization (Algorithm 2) and by "ST" a default random initialization (i.e., from a complete binary tree of depth Δ and random node parameters with Gaussian (0,1) and normalized to unit length). Additionally, ST$^+$ shows the results of using cross-entropy loss with $\beta = 100$ (see section 4.2).

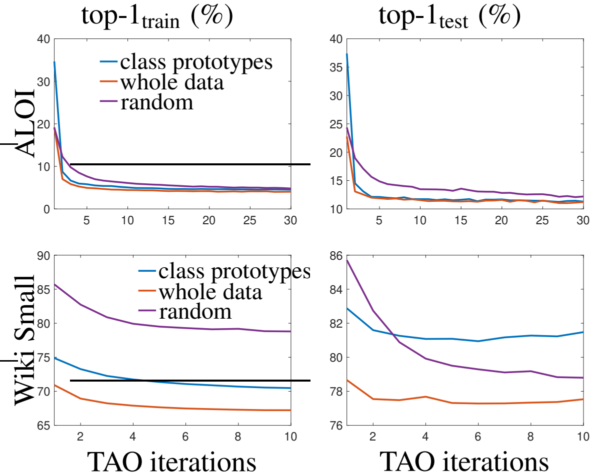

Figure 3: Comparison of different initialization methods in ST. "random" refers to the initialization with a complete binary tree of depth Δ and random node parameters. "class prototypes" refers to the initialization based on hierarchical clustering described in the Algorithm 2. "whole data" is a slight variation of it, where instead performing $\mathcal{K}$-means on the class prototypes, it does that on the whole dataset to obtain initial leaf clusters. Results show that clustering-based initializations can considerably boost the performance.

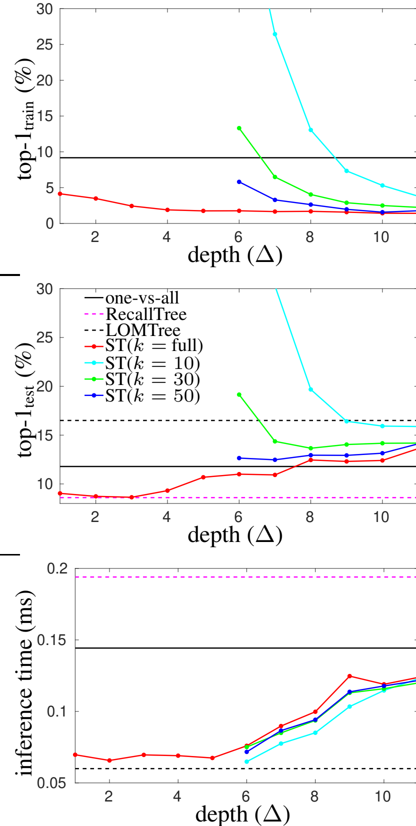

Figure 4: Error and inference time tradeoff: similar to fig. 2 but for ALOI. Here, "k = full" means we use the set of all available classes at each leaf (i.e. do not choose top-k classes).

	Method	top-1$_{\text{train}}$ (%)	top-1$_{\text{test}}$ (%)	Δ	# leaves	# classes per leaf	Inference Time per example (ms)
ALOI	MACH ($B = 512$,$R = 50$)	22.22	24.63	–	–	–	3.13
	LOMTree	–	16.50 ±0.70	10	–	(10)	(0.06)
	ST($k = 30$)	3.78 ±0.27	12.51 ±0.24	8	256	26	0.09
	ST($k = 50$)	3.57 ±	12.01 ±	7	128	41	0.09
	one-vs-all	9.17 ±0.08	11.91 ±0.24	0	1	1000	0.14
	ST^{+}($k = 30$)	4.15 ±0.06	11.22 ±0.09	8	229	24.5	0.02
	ST($k =$ full)	1.65 ±0.05	10.78 ±0.31	7	106	56	0.08
	ST†($k = 30$)	4.10 ±0.00	10.47 ±0.00	8	246	20	0.01
	RecallTree	4.37	9.90	12	–	58	0.19
	ST†($k = 100$)	2.54 ±0.03	9.33 ±0.20	7	8	43.6	0.02
	ST($k =$ full)	2.37 ±0.08	8.54 ±0.27	3	8	436	0.06
WIKI–Small	RecallTree	87.34	92.64	15	–	60	0.97
	one-vs-all	49.62	85.71	0	1	36k	10.70
	MACH ($B = 32$,$R = 25$)	53.98	84.80	–	–	–	252.64
	ST($k = 200$)	65.00	84.74	8	256.00	165.5	0.62
	ST($k = 50$)	65.76	81.02	8	256.00	50.00	0.51
	ST($k = 80$)	66.66	80.80	7	128.00	80.00	0.34
	ST($k = 100$)	71.86 ±0.16	79.68 ±0.24	7	71.67	87.82	0.36
	(π, κ)-DS	–	78.02	–	–	–	10.33
	ST†($k = 100$)	68.42	77.26	7	117.00	95.00	0.33
	ST†($k = 300$)	67.58	76.86	7	114	253	0.49
	ST†($k = 150$)	66.95	76.33	8	215	126	0.57
	ST$^{+\dagger}$($k = 150$)	65.04 ±0.09	75.62 ±0.04	8	143	140	0.52
ODP	RecallTree	93.12	94.64	6	–	400	8.42
	LOMTree	–	93.46 ±0.12	17	–	(17)	(0.26)
	one-vs-all	64.24	89.22	0	1	105k	1317.58
	ST($k = 50$)	71.98 ± 0.28	88.44 ±0.33	11	2432.00	39.70	10.92
	ST($k = 100$)	70.52 ± 0.23	88.29 ±0.30	11	1043.33	70.92	11.54
	ST($k = 1000$)	63.67 ± 0.25	86.54 ±0.29	8	209.00	684.12	10.23
	MACH ($B = 32$,$R = 25$)	46.38	84.55	–	–	–	684.04
	ST†($k = 300$)	62.86	83.78	9	512.00	129.87	9.59
	ST$^{+\dagger}$($k = 300$)	51.37	81.84	9	380.00	196.20	9.87

Table 9: Results on text classification datasets (sorted by decreasing test error): similar to Table 2 but with additional results on ALOI. Moreover, we report the train error, std over 3 independent runs (when applicable), average number of leaves of a tree and average number of classes per leaf. "†" denotes a ST version with clustering-based initialization (Algorithm 2) and "+" shows the results of using cross-entropy loss with $\beta = 100$ (see section 4.2).

Method	E_{train} (%) top-1/top-5	PPL$_{\text{train}}$ (% covered)	PPL$_{\text{test}}$ smooth	#leaf	#class /leaf
HSM-apprx	91.96/85.17	557 (100%)	–	6k	1
HSM	90.92/80.21	557 (100%)	–	6k	1
one-vs-all	86.26/80.18	125 (100%)	–	1	6k
ST($k = 50$)	85.02/68.05	11 (45.2%)	869	256	26
ST($k = 100$)	85.50/66.92	17 (52.1%)	761	128	31
ST($k = 200$)	85.17/65.64	25 (59.6%)	657	64	86
ST($k = 400$)	85.58/66.63	42 (67.0%)	566	32	136
ST($k = 800$)	85.37/65.22	59 (76.5%)	427	16	357

Table 10: Results on PennTreebank (PTB)–extension of Table 3 where we additionally report the train Perplexity, train errors, total number of leaves and average number of classes per leaf. Moreover, for our ST, we additionally report the PPL score where smoothing is applied to handle zero probabilities, i.e., we assign some small epsilon to all instances with zero probability and renormalize the output distribution (see PPL$_{\text{test}}$ smooth).

Method	E_{train} (%) top-1/top-5	PPL$_{\text{train}}$ (% covered)	PPL$_{\text{test}}$ smooth
softmax	69.07 / 48.24	44.34 (100%)	–
ST($k = 50$)	69.20 / 51.93	6.07 (59.8%)	287
ST($k = 100$)	68.92 / 50.72	8.06 (65.2%)	256
ST($k = 200$)	68.71 / 49.83	10.74 (70.7%)	201
ST($k = 400$)	68.79 / 49.20	14.39 (76.2%)	178
ST($k = 800$)	68.70 / 48.57	19.08 (81.6%)	145

Table 11: Results on PennTreebank (but trained on output of the LSTM)–extension of Table 4 but the train errors/scores are additionally reported. Also, we provide PPL scores with smoothing (as in Table 3).