Quickly Boosting Decision Trees – Pruning Underachieving Features Early

Ron Appel
Caltech, Pasadena, CA 91125 USA

Thomas Fuchs
Caltech, Pasadena, CA 91125 USA

Piotr Dollár
Microsoft Research, Redmond, WA 98052 USA

Pietro Perona
Caltech, Pasadena, CA 91125 USA

Abstract

Boosted decision trees are one of the most popular and successful learning techniques used today. While exhibiting fast speeds at test time, relatively slow training makes them impractical for applications with real-time learning requirements. We propose a principled approach to overcome this drawback. We prove a bound on the error of a decision stump given its preliminary error on a subset of the training data; the bound may be used to prune unpromising features early on in the training process. We propose a fast training algorithm that exploits this bound, yielding speedups of an order of magnitude at no cost in the final performance of the classifier. Our method is not a new variant of Boosting; rather, it may be used in conjunction with existing Boosting algorithms and other sampling heuristics to achieve even greater speedups.

1. Introduction

Boosting is one of the most popular learning techniques in use today, combining many weak learners to form a single strong one (Schapire, 1990; Freund, 1995; Freund & Schapire, 1996). Shallow decision trees are commonly used as weak learners due to their simplicity and robustness in practice (Quinlan, 1996; Breiman, 1998; Ridgeway, 1999; Kotsiantis, 2007). This powerful combination of Boosting and decision trees is the learning backbone behind many state-of-the-art methods across a variety of domains such as computer vision, behavior analysis, and document ranking to name a few (Dollár et al., 2012; Burgos-Artizzu et al., 2012; Asadi & Lin, 2013), and have the added benefit of exhibiting fast speeds at test time.

Learning speed is important as well. In active or real-time learning situations such as for human-in-the-loop processes or when dealing with data streams, classifiers must learn quickly to be practical. This is our motivation: faster training without sacrificing accuracy. To this end, we propose a principled approach. Our method offers a speedup of an order of magnitude over prior approaches while maintaining identical performance. The faster the training, the more experiments can be run, the better the science.

The contributions of our work are the following:

1. We prove a bound on the misclassification error of a decision stump given its performance on a subset of the data.
2. Based on this bound, we propose an algorithm that is guaranteed to produce identical decision trees as classical algorithms but faster by one order of magnitude.
3. We outline an algorithm for quickly Boosting decision trees using our quick tree-training method, applicable to any variant of Boosting.

In the following sections, we discuss related work, inspect the tree-boosting process, describe our algorithm, and prove our bound, and we conclude with experiments on several datasets, demonstrating our gains.

2. Related Work

Many variants of Boosting (Freund & Schapire, 1996) have proven to be competitive in terms of prediction accuracy in a variety of applications (Bühlmann & Hothorn, 2007).
Quickly Boosting Decision Trees

however, the slow training speed of boosted trees remains a practical drawback. Accordingly, a large body of literature is devoted to speeding up Boosting, mostly falling into two categories: methods that subsample features or data points, and methods that speed up training of the trees themselves.

In many situations, groups of features are highly correlated. By carefully choosing exemplars, an entire set of features can be pruned based on the performance of its exemplar. (Dollár et al., 2007) propose clustering features based on their performances in previous stages of boosting. (Kégl & Busa-Fekete, 2009) partition features into many subsets, deciding which ones to inspect at each stage using adversarial multi-armed bandits. (Paul et al., 2009) use random projections to reduce the dimensionality of the data, in essence merging correlated features.

Other approaches subsample the data. In weight-trimming (Friedman, 2000), all samples with weights smaller than a certain threshold are ignored. With Stochastic Boosting (Friedman, 2002), each weak learner is trained on a random subset of the data. For very large datasets or in the case of on-line learning, elaborate sampling methods have been proposed, i.e. Hoeffding trees (Domingos & Hulten, 2000) and Filter Boost (Domingo & Watanabe, 2000; Bradley & Schapire, 2007). To this end, probabilistic bounds can be computed on the error rates given the number of samples used (Mnih et al., 2008). More recently, Laminating (Dubout & Fleuret, 2011) trades off the number of features for the number of samples considered as training progresses, allowing for Boosting in constant time.

Although the discussed methods can be made to work in practice, they provide no performance guarantees.

A third line of work focusses on speeding up the training of decision trees. Building upon the C4.5 tree-training algorithm of (Quinlan, 1993), by using shallow (depth-\(D\)) trees and quantizing features values into \(B \ll N\) bins, an efficient implementation runs in \(O(D \times K \times N)\), where \(K\) is the number of features, and \(N\) is the number of samples (Wu et al., 2008; Sharp, 2008).

Orthogonal to all of the above methods is the use of parallelization; multiple cores or graphic cards (GPUs). Recently, (Svore & Burges, 2011) distributed computation over cluster nodes for the application of ranking; however, reporting lower accuracies as a result. GPU implementations of GentleBoost exist for object detection (Coates et al., 2009) and for medical imaging using Probabilistic Boosting Trees (PBT) (Birkbeck et al., 2011). Although these methods offer speedups in their own right, we focus on the single-core paradigm, saving parallelization for later.

Regardless of which heuristics are used for subsampling, once the subset of features and data points is obtained, weak learners are trained on that subset in its entirety. Hence, each of the aforementioned strategies can be viewed as a two-stage process; in the first stage, a smaller set of features or data points is collected, and in the second stage, decision trees are trained on that entire subset.

We propose a method for speeding up this second stage; thus, our approach can be used in conjunction with all the prior work mentioned above for even greater speedups. Unlike previous methods, our approach provides a performance guarantee: the final classifier has identical classification performance to classical training.

3. Boosting Trees

A boosted classifier of the form \(H(x) = \sum_t \alpha_t h_t(x)\) can be trained by greedily minimizing a loss function \(L\); i.e. by optimizing scalar \(\alpha_t\) and weak learner \(h_t(x)\) at each iteration \(t\). Before training begins, each data sample \(x_i\) is assigned a non-negative weight \(w_i\) (which is derived from \(L\)). After each iteration, misclassified samples are weighted more heavily thereby increasing the severity of misclassifying them in following iterations. Regardless of the type of Boosting used (i.e. AdaBoost, LogitBoost, L2Boost, etc.), each iteration requires training a new weak learner given the sample weights. We focus on the case when the weak learners are shallow decision trees.

3.1. Training Decision Trees

A decision tree \(h_{\text{TREE}}(x)\) is composed of a stump \(h_j(x)\) at every non-leaf node \(j\). Trees are commonly grown using a greedy procedure as described in (Breiman et al., 1984; Quinlan, 1993), recursively setting one stump at a time, starting at the root and working through to the lower nodes. Each stump produces a binary decision; it is given an input \(x \in \mathbb{R}^K\), and is parametrized with a polarity \(p \in \{\pm 1\}\), a threshold \(\tau \in \mathbb{R}\), and a feature index \(k \in \{1, 2, ..., K\}\):

\[
h_j(x) = p_j \text{sign}(x[k] - \tau)
\]

[where \(x[k]\) indicates the \(k\)th feature (or dimension) of \(x\)]

Note that even in the multi-class case, stumps are trained in a similar fashion, with the binary decision discriminating between a subset of classes and their complement. However, this is transparent to the training routine; thus, we only discuss binary stump training.

The goal in each stage of decision stump training is to find the optimal parameters that minimize \(\varepsilon\), the weighted misclassification error:

\[
\varepsilon \equiv \frac{1}{Z} \sum w_i I(h(x_i) \neq y_i)
\]

[where \(Z \equiv \sum w_i\) and \(1_{\{\ldots\}}\) is the indicator function]

\[
\Rightarrow \varepsilon = \frac{1}{Z} \left[ \sum_{y_i = +p} w_i I(x_i[k] \leq \tau) + \sum_{y_i = -p} w_i I(x_i[k] > \tau) \right]
\]
In practice, this error is minimized by selecting the single best feature $k^*$ from all of the features:

$$
\varepsilon^{(k^*)} = \frac{1}{Z} \left[ \sum_{y_i = +p} w_i \mathbf{1}_{x_i[k] \leq \tau} + \sum_{y_i = -p} w_i \mathbf{1}_{x_i[k] > \tau} \right]
$$

$$\{p^*, \tau^*, k^*\} = \arg\min_{p, \tau, k} \varepsilon^{(k)}, \quad \varepsilon^* \equiv \varepsilon^{(k^*)}
$$

In current efficient implementations of Boosting (Sharp, 2008), feature values are quantized into $B$ bins by linearly distributing them in $[1, B]$ (outer bins corresponding to the min/max, or to a fixed number of standard deviations from the mean), or by any other quantization method. Not surprisingly, using too few bins reduces threshold precision and hence overall performance, however, we found that $B = 256$ is large enough to incur no loss in practice.

Determining the optimal threshold $\tau^*$ requires accumulating each sample’s weight into discrete bins corresponding to that sample’s feature value $x_i[k]$. This procedure turns out to still be quite costly: for each of the $K$ features, the weight of each of the $N$ samples has to be added to a bin, making the stump training operation $O(K \times N)$ – the very bottleneck of training boosted trees.

In the following sections, we examine the stump-training process in greater detail and develop an intuition for how we can reduce computational costs.

### 3.2. Progressively Increasing Subsets

Let us assume that at the start of each Boosting iteration, the data samples are sorted in order of decreasing weight, i.e: $w_i \geq w_j \forall i < j$. Consequently, we can define the heaviest subset of data points $S_m$ and its mass $Z_m$ as:

$$S_m \equiv \{w_i\}_{i=1}^{m}, \quad Z_m \equiv \sum_{i \in S_m} w_i \quad \text{[note: } Z_N \equiv Z\]}

Clearly, $Z_m$ is greater or equal to the sum of any $m$ other sample weights. Hence, as we increase $m$, the $m$-subset includes more samples, and accordingly, its mass $Z_m$ increases (although by a diminishing rate).

In Figure 1, we plot $Z_m/Z$ for progressively increasing $m$-subsets, averaged over multiple iterations. An interesting empirical observation can be made about Boosting: a large fraction of the overall weight is accounted for by just a few samples. Since training time is dependent on the number of samples and not on their cumulative weight, we should be able to leverage this and train using only a subset of the data. The more non-uniform the weight distribution, the more we can leverage; hence, Boosting is an ideal situation: few samples are responsible for most of the weight.

The same observation was made by (Friedman et al., 2000), giving rise to the idea of weight-trimming, which proceeds as follows. At the start of each Boosting iteration, the samples are sorted in order of weight (from largest to smallest). For that iteration, the only samples used for training are those in the smallest $m$-subset such that $Z_m/Z \geq \eta$ for some predefined threshold $\eta$. All the other samples are temporarily ignored – or trimmed.

Friedman et al. claimed this to “dramatically reduce computation for boosted models without sacrificing accuracy.” In particular, they prescribed $\eta = 90\%$ to $99\%$ “typically”, but they left open the question of how to choose $\eta$ from the statistics of the data (Friedman et al., 2000).

Their work raises an interesting question: Is there a principled way to determine (and train on only) the smallest subset after which including more samples does not alter the final stump? Indeed, we can prove that a relatively small $m$-subset contains enough information to set the optimal stump, and thereby save a lot of computation.

### 3.3. Preliminary Errors

Given a feature $k$, we define the best preliminary error $\varepsilon^{(k)}$ as the lowest achievable error if only the heaviest $m$-subset of data points existed (i.e. if all other data points had zero-weight).

$$
\varepsilon_m^{(k)} = \frac{1}{Z_m} \left[ \sum_{i \in S_m} w_i \mathbf{1}_{x_i[k] \leq \tau_m^{(k)}} + \sum_{i \in S_m} w_i \mathbf{1}_{x_i[k] > \tau_m^{(k)}} \right]
$$

[where $p_m^{(k)}$ and $\tau_m^{(k)}$ are the corresponding optimal parameters]

The emphasis on preliminary indicates that the subset does not contain all data points, i.e: $m < N$, and the emphasis on best indicates that no choice of polarity $p$ or threshold $\tau$ can lead to a lower preliminary error using that feature.

As previously described, given a feature $k$, a stump is trained by accumulating sample weights into bins. We can...
view this as a progression; initially, only a few samples are binned, and as training continues, more and more samples are accounted for; hence, $\varepsilon_m^{(k)}$ may be computed incrementally which is evident in the smoothness of the traces.

Figure 2. Traces of the best preliminary errors over increasing $Z_m/Z$. Each curve corresponds to a different feature. The dashed orange curve corresponds to a feature that turns out to be misleading (i.e. its final error drastically worsens when all the sample weights are accumulated). The thick green curve corresponds to the optimal feature; note that it is among the best performing features even when training with relatively few samples.

Figure 2 shows the best preliminary error for each of the features in a typical experiment. By examining Figure 2, we can make four observations:

1. As expected, most of the overall weight is accounted for by the first few samples. (This can be seen by comparing the top and bottom axes in the figure.)
2. Feature errors become increasingly stable as $m \to N$.
3. The best performing features at smaller $m$-subsets (i.e. $Z_m/Z < 80\%$) can end up quite poorly once all of the weights are accumulated (dashed orange curve).
4. The optimal feature is among the best features for smaller $m$-subsets as well (solid green curve).

From the full training run shown in Figure 2, we note (in retrospect) that having used an $m$-subset where $m \approx 0.2N$ would have sufficed in determining the optimal feature. But how can we know a priori which $m$-subset is good enough?

Averaging over many training iterations, in Figure 3, we plot the probability that the optimal feature is among the top-performing features when trained on only an $m$-subset of the data. This gives us an idea as to how small $m$ can be while still correctly predicting the optimal feature.

From Figure 3, we see that the optimal feature is not amongst the top performing features until a large enough $m$-subset is used – in this case, $Z_m/Z \approx 75\%$. Although “optimality” is not quite appropriate to use in the context of greedy stage-wise procedures (such as boosted trees), consistently choosing sub-optimal parameters at each stump empirically leads to substantially poorer performance, and should be avoided. In the following section, we outline our approach which determines the optimal stump parameters using the smallest possible $m$-subset.

4. Pruning Underachieving Features

Figure 3 suggests that the optimal feature can often be estimated at a fraction of the computational cost using the following heuristic:

<table>
<thead>
<tr>
<th>Faulty Stump Training</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Test each feature only on samples in the $m$-subset where $Z_m/Z \approx 75%$.</td>
</tr>
<tr>
<td>2. Prune all but the 10 best performing features.</td>
</tr>
<tr>
<td>3. For each of the un-pruned features, complete training on the full data set.</td>
</tr>
<tr>
<td>4. Finally, report the best performing feature (and corresponding parameters).</td>
</tr>
</tbody>
</table>

This heuristic does not guarantee to return the optimal feature, since premature pruning can occur in step 2. However, if we were somehow able to bound the error, we would be able to prune features that would provably underachieve (i.e. would have no more chance of being optimal).

**Definition:** if a feature $k$ is guaranteed to perform worse than the best-so-far feature $k'$, then it is **underachieving**.

**Proposition 1:** for a feature $k$, the following bound holds (proof given in Section 4.2): given two subsets (where one is larger than the other), the product of subset mass and preliminary error is always greater for the larger subset:

$$S_m \subseteq S_n \Rightarrow Z_n \varepsilon_n^{(k)} \geq Z_m \varepsilon_m^{(k)}$$ (1)
Let us assume that the best-so-far error $\varepsilon'$ has been determined over a few of the features (and the parameters that led to this error have been stored). Hence, this is an upper-bound for the error of the stump currently being trained. For the next feature in the queue, even after a smaller $m$-subset $S_m \subset N$, we can guarantee that:

$$Z_m\varepsilon_m^{(k)} \geq Z\varepsilon' \Rightarrow Z\varepsilon^{(k)} \geq Z\varepsilon' \Rightarrow \varepsilon^{(k)} \geq \varepsilon'$$

Therefore, if: $Z_m\varepsilon_m^{(k)} \geq Z\varepsilon'$ then feature $k$ is underachieving and can be safely pruned. Note that the lower the best-so-far error $\varepsilon'$, the harsher the bound; consequently, it is desirable to train a relatively low-error feature early on.

Accordingly, we propose a new method based on comparing feature performance on subsets of data, and consequently, pruning underachieving features:

### Quick Stump Training

1. Test each feature only on data in a relatively small $m$-subset.
2. Sort the features based on their preliminary errors (from best to worst).
3. For each feature, continue training on progressively larger subsets and update preliminary error.
   - if it proves to be underachieving, prune it immediately.
   - if it trains to completion, save it as best-so-far.
4. Finally, report the best performing feature (and corresponding parameters).

### 4.1. Subset Scheduling

Deciding which schedule of $m$-subsets to use is a subtlety that requires further explanation. Although this choice does not affect the optimality of the trained stump, it may affect speedup. If the first "relatively small" $m$-subset (as prescribed in step 1) is too small, we may lose out on low-error features leading to less-harsh pruning. If it is too large, we may be doing unnecessary computation. Furthermore, since the calculation of preliminary error does incur some (albeit, low) computational cost, it is impractical to use every $m$-subset on progressively larger subsets.

To address this issue, we implement a simple schedule: The first $m$-subset is determined by the parameter $\eta_{kp}$ such that $Z_m/Z \approx \eta_{kp}$. $M$ following subsets are equally spaced out between $\eta_{kp}$ and 1. Figure 4 shows a parameter sweep over $\eta_{kp}$ and $M$, from which we fix $\eta_{kp} = 90\%$ and $M = 20$, and use this setting for all of our experiments.

Using our quick stump training procedure, we determine the optimal parameters without having to consider every sample for each feature. By pruning underachieving features, a lot of computation is saved. We now outline the full Boosting procedure using our quick training method:

#### Quick Stump Training

1. Initialize sample weights (sorted in decreasing order).
2. Train decision tree $h_t$ (one node at a time) using the Quick Stump Training method.
3. Perform standard Boosting steps:
   - (a) determine optimal $\alpha_t$ (using line-search).
   - (b) update sample weights given the misclassification error of $h_t$ and the specific variant of Boosting.
   - (c) if more Boosting iterations are needed, sort sample weights in decreasing order, increment iteration number $t$, and goto step 2.

We note that sorting the weights in step 3(c) above is an $O(N)$ operation. Given an initially sorted set, Boosting updates the sample weights based on whether the samples were correctly classified or not. All correctly classified samples are weighted down, but they maintain their respective ordering. Similarly, all misclassified samples are weighted up, also maintaining their respective ordering. Finally, these two sorted lists are merged in $O(N)$.

We now give a proof for the bound that our method is based on, and in the following section, we demonstrate its effectiveness in practice.

### 4.2. Proof of Proposition 1

As previously defined, $\varepsilon_m^{(k)}$ is the preliminary weighted misclassification error computed on samples in $S_m$ using the feature $k$, hence:

$$Z_m\varepsilon_m^{(k)} = \sum_{i \in S_m} w_i 1_{\{x_i[k] \leq \tau^{(k)}_m\}} + \sum_{i \in S_m} w_i 1_{\{x_i[k] > \tau^{(k)}_m\}}$$

**Proposition 1:** $S_m \subseteq S_n \Rightarrow Z_m\varepsilon_m^{(k)} \geq Z_n\varepsilon_n^{(k)}$

**Proof:** since $\varepsilon_m^{(k)}$ is the best achievable preliminary error on subset $S_m$ (and $\{p_m\}$ are the corresponding best parameters), no other parameters can do better:

$$Z_m\varepsilon_m^{(k)} \leq \sum_{i \in S_m} w_i 1_{\{x_i[k] \leq \tau\}} + \sum_{i \in S_m} w_i 1_{\{x_i[k] > \tau\}} \forall p, \tau$$

**Figure 4.** The computational cost of training boosted trees over a range of $\eta_{kp}$ and $M$, averaged over several types of runs (with varying numbers and depths of trees). Red corresponds to higher cost, blue to lower cost. The setting that leads to the lowest computational cost is $\eta_{kp} = 90\%$ and $M = 20$. 

Quickly Boosting Decision Trees

| 1. Initialize sample weights (sorted in decreasing order). |
| 2. Train decision tree $h_t$ (one node at a time) using the Quick Stump Training method. |
| 3. Perform standard Boosting steps: |
| (a) determine optimal $\alpha_t$ (using line-search). |
| (b) update sample weights given the misclassification error of $h_t$ and the specific variant of Boosting. |
| (c) if more Boosting iterations are needed, sort sample weights in decreasing order, increment iteration number $t$, and goto step 2. |

| Quick Stump Training |
| 1. Test each feature only on data in a relatively small $m$-subset. |
| 2. Sort the features based on their preliminary errors (from best to worst). |
| 3. For each feature, continue training on progressively larger subsets and update preliminary error. |
| - if it proves to be underachieving, prune it immediately. |
| - if it trains to completion, save it as best-so-far. |
| 4. Finally, report the best performing feature (and corresponding parameters). |

| Quick Boosting Decision Trees |
| 1. Initialize sample weights (sorted in decreasing order). |
| 2. Train decision tree $h_t$ (one node at a time) using the Quick Stump Training method. |
| 3. Perform standard Boosting steps: |
| (a) determine optimal $\alpha_t$ (using line-search). |
| (b) update sample weights given the misclassification error of $h_t$ and the specific variant of Boosting. |
| (c) if more Boosting iterations are needed, sort sample weights in decreasing order, increment iteration number $t$, and goto step 2. |
Therefore, switching the optimal parameters \( \{p_m^{(k)}, \tau_m^{(k)}\} \) for potentially sub-optimal ones \( \{p_n^{(k)}, \tau_n^{(k)}\} \) (note the subtle change in indices):

\[
Z_m^{Z_m^{(k)}} \leq \sum_{i \in S_m} w_i \mathbf{1}_{\{x_i, [k] \leq \tau_m^{(k)}\}} + \sum_{i \in S_m} w_i \mathbf{1}_{\{x_i, [k] > \tau_m^{(k)}\}}
\]

Further, by summing over a larger subset \( (S_n \supseteq S_m) \), the result can only increase:

\[
Z_m^{Z_m^{(k)}} \leq \sum_{i \in S_n} w_i \mathbf{1}_{\{x_i, [k] \leq \tau_n^{(k)}\}} + \sum_{i \in S_n} w_i \mathbf{1}_{\{x_i, [k] > \tau_n^{(k)}\}}
\]

Where the right-hand side is equivalent to \( Z_n^{(k)} \); hence:

\[
Z_m^{Z_m^{(k)}} \leq Z_n^{(k)} \quad \text{Q.E.D.}
\]

5. Experiments

In the previous section, we proposed an efficient stump training algorithm and showed that it has a lower expected computational cost than the traditional method. In this section, we describe experiments that are designed to assess whether the method if practical, and whether it delivers significant training speedup. We train and test on three real-world datasets and empirically compare the speedups.

5.1. Datasets

We trained Ada-Boosted ensembles of shallow decision trees of various depths, on the following three datasets:

1. CMU-MIT Faces dataset (Rowley et al., 1996); 8.510^3 training and 4.0⋅10^3 test samples, 4,310^3 features used are the result of convolutions with Viola/Jones Haar-like wavelets (Viola & Jones, 2004). The trained binary classifier is comprised of 2000 stumps as in (Viola & Jones, 2004).

2. INRIA Pedestrian dataset (Dalal & Triggs, 2005); 1.7⋅10^4 training and 1.1⋅10^4 test samples, 5,10^3 features used are Integral Channel Features (Dollár et al., 2009). The trained binary classifier is comprised of 4000 depth-2 trees as in (Dollár et al., 2009).

3. MNIST Digits (LeCun & Cortes, 1998); 6.0⋅10^4 training and 1.0⋅10^4 test samples, 7.8⋅10^2 features used are grayscale pixel values. The trained multiclass (10-class) classifier is comprised of 1000 depth-4 trees similar to (Kégl & Busa-Fekete, 2009).

5.2. Comparisons

Quick Boosting can be used in conjunction with all of the previously mentioned heuristics to provide further gains in training speed. Computation is reported in units proportional to Flops, since running time (in seconds) is dependent on extra compiler optimizations, which are beyond the scope of this work. In Figure 5, we plot the computation cost versus training loss and versus test error.

We compare vanilla (no heuristics) AdaBoost, Weight-Trimming with \( \eta = 90\% \) and 99\% [see Section 3.2], LazyBoost 90\% and 50\% (only 90\% or 50\% randomly selected features are used to train each weak learner), and StochasticBoost (only a 50\% random subset of the samples are used to train each weak learner). To these six heuristics, we apply our method to produce six “quick” versions.

We further note that our goal in these experiments is not to tweak and enhance the performance of the classifiers, but to compare the performance of the heuristics with and without our proposed method.

5.3. Results and Discussion

From Figure 5, we make several observations. “Quick” versions require less computational costs (and produce identical classifiers) as their slow counterparts. From the training loss plots (5a1, 5b1, 5c1), we gauge the speed-up offered by our method; often around an order of magnitude. Quick-LazyBoost-50\% and Quick-StochasticBoost-50\% are the least computationally-intensive heuristics, and vanilla AdaBoost always achieved the smallest training loss and attained the lowest test error in two of the three datasets.

Recall that the motivation for this work was to speed up training for two possible reasons: (a) given a fixed computational budget, the best possible classifier could be trained, and (b) given a desired performance, a classifier could be trained with the least computational cost.

For each dataset, we find the single fastest-trained heuristic and record its computational cost. Given that cost as our budget, we compare the test errors achieved by all of the heuristics (with and without our method) and plot the relative gains in Figure 6. For most of the heuristics, there is a two to eight-fold reduction in test error, whereas for weight-trimming, we see less of a benefit. In fact, for the second dataset, Weight-trimming-90\% runs at the same cost with and without our speedup.

Similarly, in Figure 7, we compare how much less computation is required to achieve the best error rate using our method for each heuristic. Most heuristics see an eight to sixteen-fold reduction in computational cost, whereas for weight-trimming, there is still a speedup, albeit again, a much smaller one (between one and two-fold).

As discussed in Section 3.2, weight-trimming acts similarly to our proposed method in that it prunes features, although it does so naively - without adhering to a provable bound.
Quickly Boosting Decision Trees

6. Conclusions

We presented a principled approach for speeding up training of boosted decision trees. Our approach is built on a novel bound on classification error, guaranteeing that gains in speed do not come at a loss in classification performance.

Experiments show that our method is able to reduce training time by an order of magnitude or more, or given a fixed computational budget, is able to train classifiers that reduce errors on average by two-fold or more. Our ideas may be applied concurrently with other techniques for speeding up Boosting (e.g. subsampling of large datasets) and do not limit the generality of the method. Our method is transformative; it enables the use of Boosting in applications where fast training of classifiers is key.
Acknowledgements

This work was supported by NSERC-420456-2012, MURI-ONR N00014-10-1-0933, ARO/JPL-NASA Stennis NAS7.03001, and Moore Foundation.

References


Birkbeck, N., Sofka, M., and Zhou, S. K. Fast boosting trees for classification, pose detection, and boundary detection on a GPU. In Computer Vision and Pattern Recognition Workshops (CVPRW), 2011. 2


Bühlmann, P. and Hothorn, T. Boosting algorithms: Regularization, prediction and model fitting. In Statistical Science, 2007. 1


Coates, A., Baumstarck, P., Le, Q., and Ng, A. Y. Scalable learning for object detection with GPU hardware. In Intelligent Robots and Systems (IROS), 2009. 2


Dollár, P., Tu, Z., Tao, H., and Belongie, S. Feature mining for image classification. In Computer Vision and Pattern Recognition (CVPR), June 2007. 2

Dollár, P., Tu, Z., Perona, P., and Belongie, S. Integral channel features. In British Machine Vision Conference (BMVC), 2009. 6


Domingo, C. and Watanabe, O. Scaling up a boosting-based learner via adaptive sampling. In Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2000. 2

Domingos, P. and Hulten, G. Mining high-speed data streams. In International Conference on Knowledge Discovery and Data Mining, 2000. 2


Freund, Y. Boosting a weak learning algorithm by majority. In Information and Computation, 1995. 1


Kégl, B. and Busa-Fekete, R. Accelerating adaboost using UCB. KDD-Cup 2009 competition, 2009. 2, 6


LeCun, Y. and Cortes, C. The MNIST database of handwritten digits, 1998. 6


Paul, B., Athithan, G., and Murty, M.N. Speeding up adaboost classifier with random projection. In International Conference on Advances in Pattern Recognition (ICAPR), 2009. 2


Quinlan, R. J. Bagging, boosting, and C4.5. In National Conference on Artificial Intelligence, 1996. 1


Schapire, R. E. The strength of weak learnability. In Machine Learning, 1990. 1

Sharp, T. Implementing decision trees and forests on a gpu. In European Conference on Computer Vision (ECCV), 2008. 2, 3

