International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 11, Number 2, pages 215–230, July 2021

ExtraFerns: Fully Parallel Ensemble Learning Technique with Random Projection and Non-Greedy yet Minimal Memory Access Training

Shungo Kumazawa, Kazushi Kawamura, Thiem Van Chu, Masato Motomura, and Jaehoon Yu
Tokyo Institute of Technology
Yokohama, Kanagawa, 226-8503, Japan

Received: February 15, 2021
Revised: May 3, 2021
Accepted: May 25, 2021
Communicated by Ikki Fujiwara

Abstract

Training machine learning models on edge devices is always a conflict with power consumption and computing cost. This paper introduces a hardware-oriented training method called *ExtraFerns* for a unique subset of decision tree ensembles, which significantly decreases memory access and optimizes each tree in parallel. ExtraFerns benefits from the advantages of both extraTrees and randomFerns. As extraTrees does, it generates nodes by randomly selecting attributes and generating thresholds. Then, as randomFerns does, it builds ferns, which are decision trees that share identical nodes at each depth. In contrast to other ensemble methods using greedy optimization, ExtraFerns attempts global optimization of each fern. Experimental results show that ExtraFerns requires only 4.3% and 4.1% memory access for training models with 3.0% and 1.2% accuracy drops compared with randomForest and extraTrees, respectively. This paper also proposes applying lightweight random projection to ExtraFerns as a preprocessing step, which achieved a further accuracy improvement of up to 2.0% for image datasets.

Keywords: ensemble learning, fern ensemble, decision tree ensemble, non-greedy optimization, parallel optimization, random projection

1 Introduction

Machine learning on edge devices, so-called edge AI, is a natural progression given the advent of computationally powerful and efficient IoT devices. The most typical form of edge AI is load balancing: training on clouds and inference on the edge. By reducing data transmission from an edge to clouds, edge AI provides a solution for privacy, high power consumption, and low bandwidth issues. However, edge AI often requires on-site training with the data obtained from edge devices because training on clouds requires massive data transmission from the edge to clouds, resulting in increased power consumption. Then, why is on-site training less common? What makes it challenging?

The challenge is that the computational cost for training is unaffordable to edge devices; training on the edge is not advantageous compared with training on clouds, even considering the data transmission. For example, neural networks require significant computational capabilities and large memory for their training based on back-propagation. However, it is difficult for edge devices to satisfy these requirements or to afford the power consumption required for such computation. Therefore, machine learning algorithms with more lightweight but efficient training methods are required for edge devices.

To tackle this problem, we propose an edge-oriented machine learning algorithm called *ExtraFerns* [1], which is based on conventional decision tree ensemble learning methods [2–5] that use decision trees as base

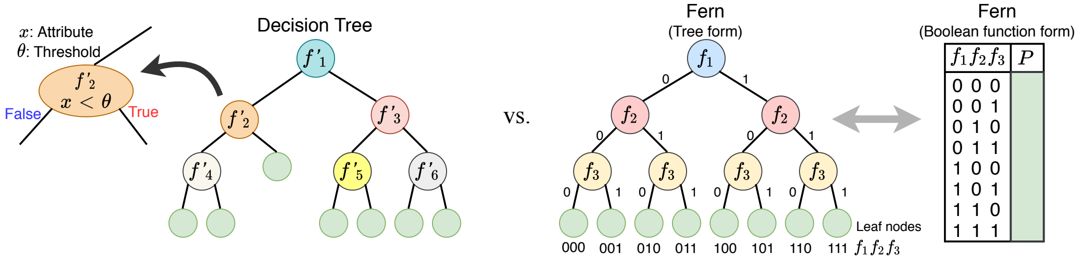

Figure 1: Decision trees and ferns. A decision tree consists of internal nodes, branches, and leaf nodes, which represent tests on attributes, the tests' outcomes, and expected values, respectively. A fern is a unique subset of decision trees. Because a fern has identical nodes at each depth, it can be considered a Boolean function.

learners: randomForest [2], extremely randomized trees (extraTrees) [3], randomFerns [4], and rFerns [5]. Decision tree ensembles are not state-of-the-art machine learning algorithms, but they still outperform neural networks for tabular data. Compared with neural networks, they require only a few parameters and a small amount of computation. These are suitable features for training on the edge, but there is still room for improvement in the training phase in terms of power consumption and processing performance.

ExtraFerns is an algorithm that takes advantage of the benefits of both extraTrees and randomFerns. As extraTrees does, it generates nodes by randomly selecting attributes and generating thresholds. This node generation largely reduces off-chip memory accesses, which are the dominant factor in power consumption. Then, as randomFerns does, it builds ferns, which are decision trees sharing identical nodes at each depth. This fern structure eliminates the processing dependency on input data. Also, by using this structure, ExtraFerns conducts non-greedy optimization in training to find optimal ferns.

The contributions of this paper can be summarized as follows:

- We describe the details of ExtraFerns, which requires an extremely small number of off-chip memory accesses.
- ExtraFerns conducts non-greedy optimization in parallel, which leads to accuracy improvement.
- We also empirically show that Dirichlet prior, as used in randomFerns, is dispensable for ExtraFerns.
- Finally, we propose applying random projection [6, 7] to ExtraFerns as a preprocessing step for further accuracy improvement.

The remainder of this paper is organized as follows. Section 2 briefly describes the features of conventional decision tree ensembles. Section 3 describes the details of the proposed ExtraFerns based on them, and then Sections 4 and 5 present preliminary experimental results and evaluation results, respectively. Section 6 describes how to introduce random projection to ExtraFerns for accuracy improvement and demonstrates its effectiveness through experimental results. Finally, Section 7 concludes this paper.

2 Related Work

Decision tree ensembles are statistical machine learning algorithms that use decision trees as base learners. As shown in Figure 1, a decision tree consists of internal nodes, branches, and leaf nodes, which represent tests on attributes, the tests' outcomes, and class labels/expected values, respectively. Ferns are decision trees with a unique structure, sharing identical internal nodes at each depth. Therefore, ferns can be considered Boolean functions using the internal nodes' outcomes as arguments.

This section briefly describes four ensemble algorithms used in designing ExtraFerns: randomForest, extraTrees, randomFerns, and rFerns. RandomForest and extraTrees use trees as base learners, while randomFerns and rFerns use ferns. Table 1 presents a complexity comparison between these decision tree ensemble

Table 1: Complexity Comparison between Ensemble Algorithms

Algorithm	# Leaf Nodes	# Internal Nodes	# Memory Accesses	Computational Complexity
randomForest	$O(M\tilde{N})$	$O(M\tilde{N})$	$O(KM\tilde{N}\log\tilde{N})$	$O(KM\tilde{N}(\log\tilde{N})^2)$
extraTrees	$O(MN)$	$O(MN)$	$O(KMN\log N)$	$O(KMN\log N)$
randomFerns	$O(2^D M)$	$O(DM)$	$O(DMN)$	$O(DMN)$
rFerns			$O(DM\tilde{N})$	$O(DM\tilde{N})$
ExtraFerns			$O(UDMN)$	$O(UDMN\log N)$

Table 2: Definitions of Variables for Complexity Comparison

Variable	Description
D	depth of trees / ferns
K	# of candidate attributes
M	# of trees / ferns
N	# of training data
$\tilde{N}$	# of bootstrapped training data
U	# of updates in ExtraFerns

algorithms in training. Table 2 defines the variables used in Table 1. The following paragraphs explain the details of each algorithm separately.

randomForest randomForest builds a forest, i.e., a set of decision trees, based on randomized node optimization [2]. It makes each tree composing the forest with $\tilde{N}$ bootstrapped samples from N training data. Once randomForest decides a training dataset to build a tree with, it randomly selects K attributes and chooses the best internal node to achieve the highest purity. In the best case, each internal node partitions a training dataset into two balanced subsets, i.e., the root node splits $\tilde{N}$ data into two sets of $\tilde{N}/2$ data; its child nodes halves the data again, etc. Therefore, in this case, the complexity of the expected depth is $O(\log\tilde{N})$, and the complexities of the number of both leaf nodes and internal nodes become $O(\tilde{N})$. The computational complexity for training a tree is $O(K\tilde{N}(\log\tilde{N})^2)$, where randomForest requires K searches for $\tilde{N}$ data for each $\log\tilde{N}$ depth because the sum of the numbers of node's data is $\tilde{N}$ at each depth, and the additional $\log\tilde{N}$ comes from the sorting procedure for optimization split thresholds [8, Sec. 5]. Table 1 lists each complexity when the size of the forest is M.

extraTrees extraTrees is a further randomized decision tree ensemble [3]. While it is similar to randomForest, it has two major differences. First, extraTrees trains trees with N training data without bootstrapping, and second, it uses randomly selected split thresholds without optimization. Therefore, extraTrees can significantly reduce the computational complexity compared with randomForest. In the best case, as shown in Table 1, each complexity, except for computational, is the same as that for randomForest except using N instead of $\tilde{N}$. The computational complexity of extraTrees for training M trees is $O(KMN\log N)$; because extraTrees does not require any sorting procedure, there is no extra $\log N$ as there is in randomForest [8, Sec. 5].

randomFerns randomFerns is a fern ensemble specialized for image classification [4]. Its unique feature is the use of two attributes in each internal node instead of using an attribute–threshold pair. This comparison can be considered binary edge extraction in image processing:

$$f = \begin{cases} 1, & \text{if } I_{x,y} > I_{x',y'} \\ 0, & \text{otherwise} \end{cases}, \tag{1}$$

where $I_{x,y}$ and $I_{x',y'}$ are pixel intensities at the (x, y) and (x', y') coordinates, respectively.

Given images from a set of C classes, $C = \{1, 2, \ldots, C\}$, and a set of outcomes from a fern with D depths, $\mathcal{F} = \{f_1, f_2, \ldots, f_D\}$, randomFerns finds the class label c^* by calculating

$$c^* = \underset{c\in C}{\arg\max}\, P(c|f_1, f_2, \ldots, f_D), \tag{2}$$

where (2) can be rewritten by Bayes' theorem as follows:

$$c^* = \underset{c\in C}{\arg\max}\, P(c)P(f_1, f_2, \ldots, f_D|c). \tag{3}$$

When using M ferns, randomFerns assumes semi-naive Bayes. Let the outcome set of the m-th fern $\mathcal{F}_m = \{f_{m1}, f_{m2}, \ldots, f_{mD}\}$. RandomFerns considers all $\mathcal{F}_m$ as independent from each other but $f_{md} \in \mathcal{F}_m$ depends on other outcomes within $\mathcal{F}_m$. This assumption leads to

$$c^* = \underset{c\in C}{\arg\max}\, P(c)\prod_{m=1}^{M} P(\mathcal{F}_m|c). \tag{4}$$

Therefore, randomFerns learns the probability distribution $P(\mathcal{F}_m|c)$ in its training phase.

The number of leaf nodes within a fern, L, is 2^D. Because this size is much larger than the size of the training data in many cases, there are many leaf nodes without assigned training data. A zero probability is not valid for (4) to get the proper class label. Therefore, randomFerns assumes a Dirichlet prior to give some baseline probability to all possible outcome sets:

$$P(\mathcal{F}_m = l|c) = \frac{N_{lc} + \epsilon}{\sum_{i=1}^{L} (N_{ic} + \epsilon)}, \tag{5}$$

where N_{lc} is the number of c-class data assigned to the l-th leaf node, and ϵ is a small positive value: $\epsilon = 1$ in [4]. Note that the binary expression of l is identical to the outcome set of $\mathcal{F}_m$.

Table 1 shows each complexity of randomFerns with M ferns. The most significant difference compared with randomForest and extraTrees is in randomFerns' complexities of memory access and computation, which are of much smaller order at $O(DMN)$.

rFerns rFerns is an extension of randomFerns for general-purpose classification. It generates ferns with randomly selected pairs of attributes and thresholds, similar to extraTrees. Because rFerns uses $\tilde{N}$ bootstrapped training data instead of N, the complexities of its memory access and computation are $O(DM\tilde{N})$, which are less than those of randomFerns.

As shown in Table 1, these methods listed above have no significant differences from each other in the amount of computation and the number of memory accesses. Most of the calculations, except for randomForest, consist of additions and comparisons, requiring one comparison operation and several addition operations to decide each branch of a decision tree from each data read. Assuming 32-bit data, the power consumption of one DRAM access is 6400x more than that of one integer addition [9]. Even if we assume the number of addition operations is 10-100x larger than the number of DRAM accesses, its power consumption is still far less than that of the DRAM access.

Most calculations of randomForest also consist of additions and comparisons but, in addition to them, it calculates the branching score multiple times proportional to the number of data read to find the optimal branching point. Each branching point calculation requires at least two multiplication for both left and right child nodes' impurity calculations. Although multiplication requires more power than addition, one DRAM access consumes 173x more energy than floating-point multiplication [9]. Even for two multiplications, the power consumption ratio is 86.5x, and memory access is still dominant for the power consumption in this case.

In these conventional methods, randomForest and extraTrees show better accuracy performance than others but require more memory accesses. That is why we mainly focus on memory access when designing a new decision tree ensemble algorithm in this paper.

3 ExtraFerns

ExtraFerns is an ensemble learning method based on extraTrees and randomFerns. This section provides an overview of ExtraFerns, details its TWO distinct key components, and analyzes its complexities.

3.1 Algorithm Overview

Figure 2 shows an overview of ExtraFerns. The learning phase consists of four processes: 1. random node generation, 2. fern construction, 3. parallel threshold optimization, and 4. leaf probability calculation. The first two processes are based on conventional methods, and the other two processes are novel approaches devised for ExtraFerns. ExtraFerns builds a fern ensemble by repeating these four processes.

In random node generation, ExtraFerns generates nodes by randomly selecting pairs of attributes and thresholds like extraTrees. Then, ExtraFerns builds a fern in the same manner as randomFerns. By adopting these processes, ExtraFerns inherits the characteristics of low memory access and low computational complexity from extraTrees and randomFerns. However, it is not a panacea because this randomized generation makes it difficult to achieve high inference accuracy: rFerns is a good example; it also adopts these processes, and its inference accuracy is significantly lower than those of other methods.

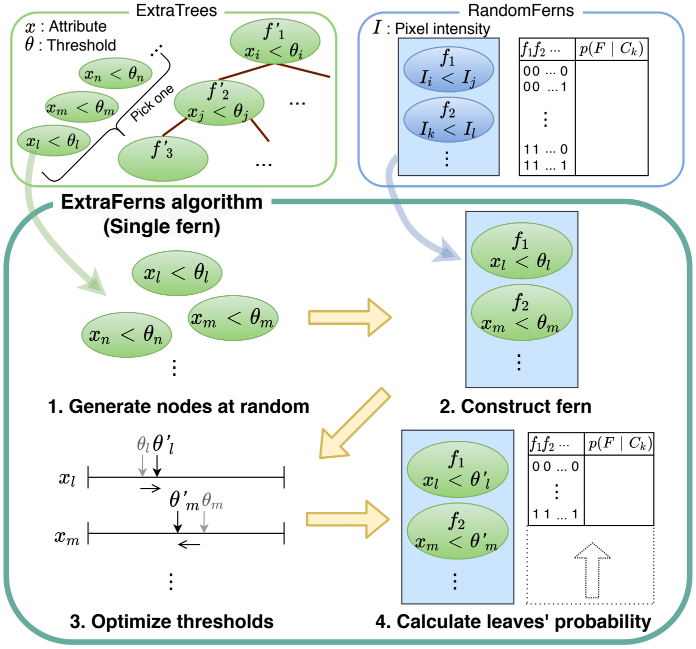

Figure 2: Overview of ExtraFerns. ExtraFerns consists of random node generation, fern construction, parallel threshold optimization, and leaf probability calculation. The former two are based on conventional methods and bring memory access and computation advantages to ExtraFerns. The latter two enable ExtraFerns to achieve comparable performance to other methods.

To solve this problem, ExtraFerns conducts parallel threshold optimization. Instead of the greedy search for optimizing each node's prediction performance, ExtraFerns searches the best thresholds for optimizing the entire fern's prediction performance. After that, ExtraFerns calculates each leaf node's probability distribution similarly to randomFerns and rFerns. The distinct difference is that ExtraFerns does not use Dirichlet priors, allowing a more than 80% reduction in memory requirements for storing leaf nodes. The following two subsections describe the details of parallel threshold optimization and leaf node probability calculation, respectively.

3.2 Parallel Threshold Optimization

Given a training dataset $\mathcal{T}$ and randomly generated fern $\mathcal{F}$, we define the objective function for threshold optimization as follows:

$$G(\Theta|\mathcal{T},\mathcal{F}) = \max_{\Theta} \sum_{l=1}^{L} N_l(\mathcal{T}) \sum_{c=1}^{C} P(c|\mathcal{F}(\Theta) = l)^2, \tag{6}$$

where Θ represents the tuple of thresholds $\{\theta_1, \theta_2, \ldots, \theta_d\}$, N_l is the number of training data assigned to the l-th leaf, and $\mathcal{F}(\Theta)$ is the fern using Θ. To achieve non-greedy optimization, in (6), we calculate the sum of each leaf's purities derived from Gini impurities. Then, we calculate the weighted sum of entire leaves' purities, where N_l is considered as a confidence factor.

For maximizing the objective function, ExtraFerns searches the best tuple of thresholds in parallel. Algorithm 1 shows the detailed process flow, where we quantize each training dataset with an 8-bit fixed-point expression in advance. As shown in Algorithm 1, parallel threshold optimization has triple-nested loops. It executes the inner two loops for each update $u \in [1, U]$. Of the two inner loops, the outer loop corresponds to the node at each depth $d \in [1, D]$, and the innermost loop corresponds to the search range $r \in [-R, +R]$. Because the iterations of the two inner loops are mutually independent, ExtraFerns executes them in parallel.

Figure 3 shows an example of parallel threshold optimization using the Iris dataset [10], where we set

Algorithm 1 Parallel Threshold Optimization

```
Input: Training dataset $\mathcal{T}$ & fern $\mathcal{F}$
Output: Tuple $\Theta^*$ of optimal thresholds for $\mathcal{F}$
$\Theta := (\theta_1, \theta_2, \ldots, \theta_D)$ s.t. $\theta_d \in [0, 255]$ $_{(d=1,2,\ldots,D)}$
for each $u \in [1, U]$ do
    for each $d \in [1, D]$ do                                  ⎫
        $\Theta' := \Theta$                                      ⎪
        $G_{max} := 0$                                           ⎪
        for each $r \in [-R, +R]$ do                             ⎪
            $\theta'_d := \theta_d + r$                          ⎪
            if $0 \le \theta'_d \le 255$ then                    ⎬ in parallel
                $\Theta'(d) := \theta'_d$                        ⎪
                if $G_{max} < G(\Theta'|\mathcal{T}, \mathcal{F})$ then   ⎪
                    $G_{max} := G(\Theta'|\mathcal{T}, \mathcal{F})$      ⎪
                    $\theta^*_d := \theta'_d$                    ⎭
    $\Theta := (\theta^*_1, \theta^*_2, \ldots, \theta^*_D)$
$\Theta^* := \Theta$
```

hyperparameters U, D, and R to 3, 4, and 11, respectively. The top row illustrates the concept of searching thresholds in parallel, and the bottom row shows the corresponding changes in the attribute space.

3.3 Sparsified Probability Distribution of Leaf Nodes

The basic equation for ExtraFerns is equivalent to (4) of randomFerns. We take the logarithm of (4) because addition is more practically efficient than multiplication. Therefore, (4) can be rewritten as

$$c^* = \operatorname*{arg\,max}_{c \in C} \sum_{m=1}^{M} \log P(\mathcal{F}_m|c) + \log P(c). \tag{7}$$

The most significant difference is in how to handle the leaves without assigned training data. While randomFerns uses Dirichlet priors, this requires a large amount of memory space for storing leaf node information. ExtraFerns reduce the memory space by modifying (5) to

$$P(\mathcal{F}_m = l|c) = \begin{cases} \frac{N_{lc}}{\sum_{i=1}^{L} N_{ic}} & (N_l \neq 0) \\ 1 & (N_l = 0) \end{cases}, \tag{8}$$

where N_l is the number of all data in the l-th leaf node. In (8), we equally assign 100% probability to every class when a leaf node is empty, ignoring the decision of an empty leaf node. Because the logarithm of one is zero, ExtraFerns' leaf information becomes sparse.

3.4 Complexity of ExtraFerns

Referring to Table 1, while ExtraFerns has the same complexities in leaf nodes and internal nodes as randomFerns and rFerns, its number of non-empty leaf nodes is much smaller compared with randomFerns and rFerns thanks to the sparsified leaf information. Due to the newly-introduced hyperparameters U and R, ExtraFerns' memory access and computational complexities are slightly larger than those of conventional methods. Because R is usually a small constant value from 5 to 10, we exclude R from the computational complexity, defining it as $O(UDMN \log N)$. The additional $\log N$ comes from the process of storing sparsified leaf nodes.

The computational complexity of ExtraFerns is not much different from that of randomForests and extraTrees, and the training time is about the same as for conventional decision tree ensembles, e.g., a few seconds or minutes to run on the CPU. This time is much less than the training time for deep neural networks, which requires a few hours or days to run on the GPU, which is one of the reasons why decision tree ensembles are suitable for training on edge.

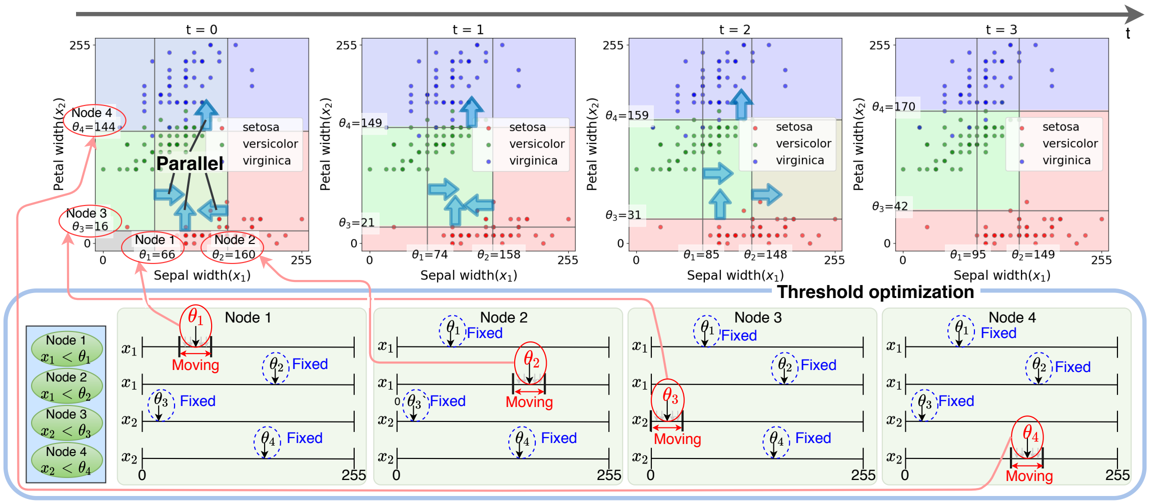

Figure 3: Example of parallel threshold optimization. The top row illustrates the concept of searching thresholds in parallel, and the bottom row shows the corresponding changes in the attribute space. Hyperparameters U, D, and R are set to 3, 4, and 11, respectively.

Table 3: Evaluation Dataset Description

Dataset	# Training Data	# Test Data	# Attributes	# Classes
MNIST	60,000	10,000	784	10
Fashion-MNIST [11]	60,000	10,000	784	10
Kuzushiji-MNIST [12]	60,000	10,000	784	10
Devanagari-Script [13]	76,666	15,334	1,024	46
HIGGS [14]	80,000	20,000	28	2
SUSY [14]	80,000	20,000	18	2

4 Preliminary Experiments

Before evaluation, we must find appropriate hyperparameters in advance because they impact the threshold optimization results. It is also necessary to verify how the absence of both bootstrapping and the Dirichlet prior influences inference accuracy. We conducted a grid search of hyperparameters and examined the accuracy of ExtraFerns by adding bootstrapping and the Dirichlet prior. Table 3 describes the datasets used in this study, where HIGGS and SUSY are randomly subsampled datasets of the originals.

4.1 Hyperparameter Grid Search

As mentioned above, ExtraFerns newly introduces two hyperparameters R and U, where R is the range of the threshold search, and U is the number of updates. If either R or U is too small, ExtraFerns falls into one of the local optimal solutions. To find appropriate R and U, we conducted a grid search on datasets in Table 3. Figure 4 shows the grid search results for accuracy in the form of heatmaps, where we set M and D to 500 and 15, respectively. As shown in Figure 4, each heatmap shows a similar tendency, and based on these results, we set R and U to 8 and 30, respectively.

4.2 Influences of Bootstrapping and Dirichlet Prior

We designed ExtraFerns without bootstrapping and Dirichlet priors based on empirical results. Table 4 lists the results comparing the baseline mode and other variations with bootstrapping and Dirichlet priors, where

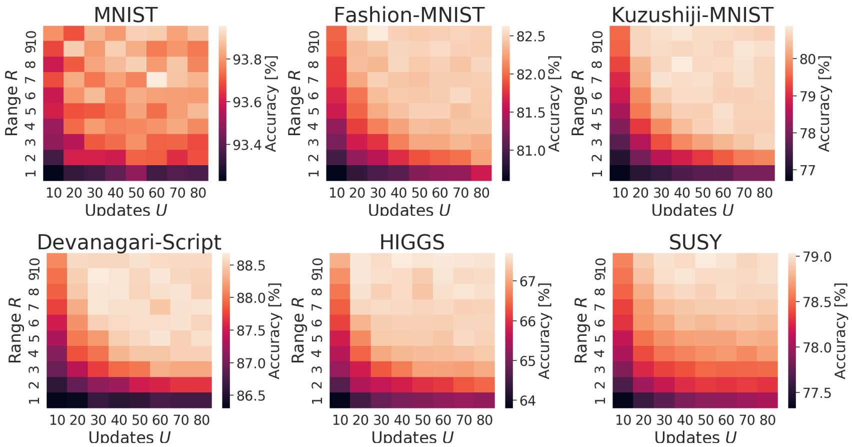

Figure 4: Heatmaps of hyperparameter grid search. Two hyperparameters R and U are examined on datasets in Table 3, and each heatmap shows a similar tendency. Based on the results, we use $R = 8$ and $U = 30$ as the parameters for other evaluation.

Table 4: Preliminary Evaluation of ExtraFerns Variations

Dataset	Accuracy[%] (diff)			
	ExtraFerns baseline	ExtraFerns +bootstrap	ExtraFerns+Dirichlet	
			$\epsilon = 0.1$	$\epsilon = 1$
MNIST	**93.9**	93.5 (−0.4)	92.7 (−1.2)	92.0 (−1.9)
Fashion-MNIST	**82.5**	80.6 (−1.9)	80.8 (−1.7)	80.2 (−2.3)
Kuzushiji-MNIST	**80.9**	78.8 (−2.1)	77.1 (−3.8)	74.5 (−6.4)
Devanagari-Script	**88.9**	86.4 (−2.5)	85.2 (−3.7)	82.1 (−6.8)
HIGGS	68.0	66.7 (−1.3)	**68.1** (+0.1)	**68.1** (+0.1)
SUSY	**78.9**	78.7 (−0.2)	78.8 (−0.1)	78.7 (−0.2)

we set M and D to 1,000 and 15, respectively. From Table 4, we can confirm that the baseline outperforms both bootstrapping variation and Dirichlet variation except in one case. Based on these results, we excluded both processes from ExtraFerns.

5 Evaluation

We compared ExtraFerns with conventional decision tree ensembles in terms of accuracy, memory space, memory access, and power consumption where randomForest, extraTrees, and rFerns were comparison targets; we excluded randomFerns because it is not applicable to general tabular data. Each implementation came from machine learning libraries: randomForest and extraTrees from scikit-learn (v0.22.1) and rFerns (v3.0.0) from R package. In the experiments, we set the number of trees/ferns to 1,000, and other values to their default. Each reported result is an average of ten trials.

5.1 Accuracy

Table 5 lists the results for accuracy of each method, where we set D to 15, 20, and 25. However, we measured the accuracy of rFerns only for D set to 15 because the implementation in the R package of rFerns only allows a depth of up to 17. It seems that this is due to the large memory requirement of rFerns because it stores all 2^D leaf nodes.

First, we compare each method for depth 15. The average accuracy of ExtraFerns was 9.3% higher than that of rFerns thanks to the threshold optimization. However, it was 3.0% and 1.2% lower than those of randomForest and extraTrees, respectively, due to the model simplicity.

Table 5: Accuracy Comparison between Ensemble Methods

Dataset	D (Depth)	Accuracy[%] (diff)			
		ExtraFerns	randomForest	extraTrees	rFerns
MNIST	15	93.9	96.8	96.7	**86.9** (−8.9)
	20	95.2	**97.1** (+1.3)	97.3	N/A
	25	**95.8**	**97.1** (+1.3)	**97.4** (+1.6)	
Fashion MNIST	15	82.5	87.1	86.3	**74.2** (−10.2)
	20	83.9	87.7	87.4	N/A
	25	**84.4**	**87.9** (+3.5)	**87.7** (+3.3)	
Kuzushiji MNIST	15	80.9	84.9	82.3	**59.3** (−24.9)
	20	83.4	86.4	86.1	N/A
	25	**84.2**	**86.5** (+2.3)	**87.1** (+2.9)	
Devanagari Script	15	88.9	89.6	87.5	**70.1** (−20.0)
	20	90.0	92.0	92.0	N/A
	25	**90.1**	**92.2** (+2.1)	**92.6** (+2.5)	
HIGGS	15	**68.0**	72.5	68.5	**68.8** (+0.8)
	20	67.2	**72.8** (+4.8)	70.3	N/A
	25	66.8	**72.8** (+4.8)	**71.1** (+3.1)	
SUSY	15	78.9	**80.0** (+0.8)	79.2	**78.1** (−1.1)
	20	**79.2**	**80.0** (+0.8)	79.7	N/A
	25	79.1	**80.0** (+0.8)	**80.0** (+0.8)	

Table 6: Comparison of Average Numbers of Leaf Nodes per Tree

Dataset	D (Depth)	# Leaf Nodes L			
		ExtraFerns	randomForest	extraTrees	rFerns
MNIST	15	539	3,803	5,590	32,768
	20	1,701	4,799	9,205	N/A
	25	4,356	4,942	10,041	
Fashion MNIST	15	3,010	2,599	4,378	32,768
	20	8,730	4,237	8,714	N/A
	25	16,042	4,771	10,723	
Kuzushiji MNIST	15	6,520	4,421	5,371	32,768
	20	18,387	6,460	10,517	N/A
	25	30,684	6,754	12,733	
Devanagari Script	15	3,371	7,733	10,007	32,768
	20	15,666	14,615	23,690	N/A
	25	35,510	15,931	28,500	
HIGGS	15	5,337	3,884	2,935	32,768
	20	26,844	8,393	10,348	N/A
	25	52,805	10,623	21,011	
SUSY	15	1,388	2,818	2,292	32,768
	20	6,605	5,921	7,969	N/A
	25	16,089	7,990	16,602	

Table 7: Memory Space Requirements for Internal and Leaf Nodes

Algorithm	Internal Nodes [byte]	Leaf Nodes [byte]
randomForest	$(L-1)(S_x + S_{\theta_f} + S_{\text{ptr}})$	LCS_P
extraTrees		
rFerns	$D(S_x + S_{\theta_f})$	
ExtraFerns	$D(S_x + S_{\theta_i})$	$L(CS_P + 2S_{\text{ptr}} + S_{\text{index}})$

* refer to Table 6 for L

Table 8: Byte Size of Each Constant

Constant	Description	Byte
S_x	size of attribute	2
S_{θ_f}	size of float threshold	4
S_{θ_i}	size of integer threshold	1
S_{ptr}	size of node address	2
S_{index}	size of index	2 or 4
S_P	size of leaf's probabilities	4

Second, we compare the accuracies for the D with the highest accuracy for each method. The shaded areas in the table are the highest accuracy for each method. ExtraFerns significantly outperformed rFerns, except on one dataset. In two of the six datasets, ExtraFerns was at least 20% more accurate than rFerns. ExtraFerns was inferior to randomForest and extraTrees in all cases. However, the accuracy difference was always within 5%, and the average differences were 2.5% and 2.4% for randomForest and extraTrees, respectively.

5.2 Memory Space

For comparison, we consider only the dominant factors occupying memory space in each method: internal nodes and leaf nodes. Table 6 lists the number of leaf nodes per tree for each method. Shaded areas indicate the highest accuracy of each method in Table 5. In Table 6, ExtraFerns shows smaller leaf node sizes than rFerns except for Devanagari-Script because there is no need to consider Dirichlet priors with ExtraFerns. Compared with randomForest and extraTrees, however, more leaf nodes were required in most cases.

We estimate the memory space requirements for each method based on Tables 7 and 8. Table 7 lists the formulae to calculate the memory sizes for internal and leaf nodes, and Table 8 lists the byte size of each constant. For internal nodes, each node includes a pair of an S_x-byte attribute x and an $S_{\theta_{f\,/\,i}}$-byte threshold θ. In contrast to other methods using 4-byte float thresholds, ExtraFerns adopts 1-byte integer thresholds. Also, randomForest and extraTrees require an S_{ptr}-byte pointer to each child node. For leaf nodes, each node includes a probability distribution for each class. Because ExtraFerns' implementation stores sparsified leaf nodes in an associative container, its leaf nodes include two S_{ptr}-byte pointers and an S_{index}-byte index. When the leaf node size is less than or equal to 2^{16}, we use a 2-byte integer for each index. When it is more than 2^{16}, we use a 4-byte integer instead.

Figure 5 depicts the data sizes of internal and leaf nodes per tree when D is 15 or 25. We excluded rFerns results from all graphs because the number of leaf nodes is too large to place on the graphs. As shown in Figure 5, while ExtraFerns requires a smaller or equivalent memory size for depth 15 compared with conventional methods, it requires more memory on average for depth 25. This large memory requirement is

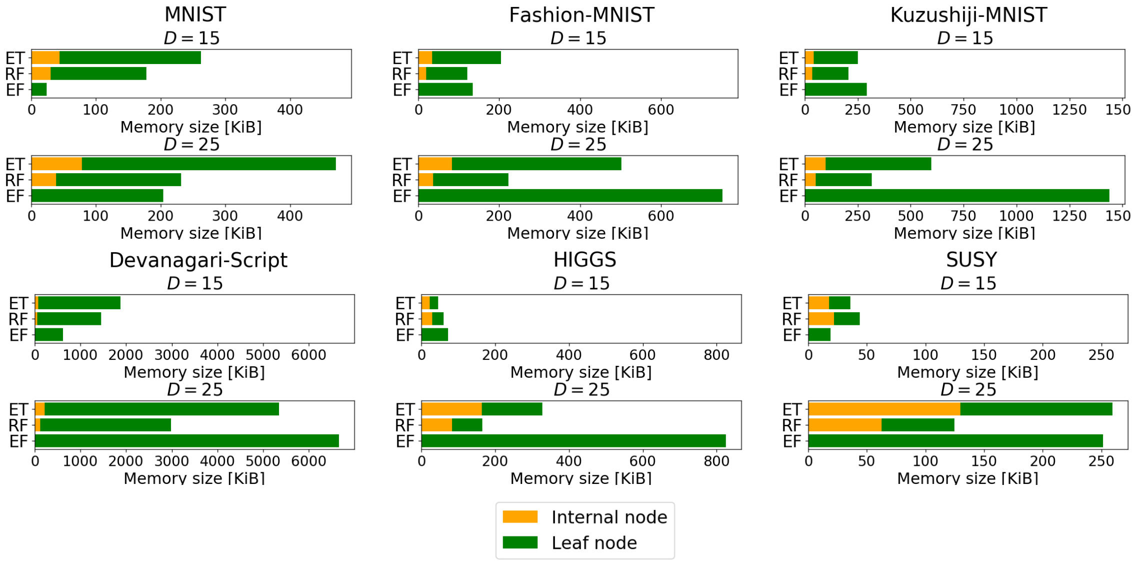

Figure 5: Memory requirement breakdown of each ensemble method. ET, RF, and EF represent extraTrees, randomForest, and ExtraFerns, respectively. When the depth is 15, ExtraFerns requires an equivalent or smaller memory size to other methods, but when the depth is 25, ExtraFerns requires more memory space than others due to the large number of leaf nodes.

a weakness of the current ExtraFerns.

5.3 Memory Access

We estimate the number of fetched training data from off-chip memory as off-chip memory accesses for the training of each method based on the following three cases: 1. small N (number of training data) and few attributes, 2. small N and many attributes, and 3. large N.

1. Small N and Few Attributes

This case corresponds to HIGGS and SUSY in Table 3. Because it is possible to put all the training data in the on-chip memory, the number of off-chip memory accesses is equivalent between all methods by storing the data in on-chip memory at the beginning of the training.

2. Small N and Many Attributes

This case corresponds to all datasets except HIGGS and SUSY in Table 3. In this case, we estimate that the number of fetched data from off-chip memory for ExtraFerns and rFerns is DN per tree because each of D nodes has one splitting attribute. ExtraFerns must store the DN samples in on-chip memory and use them every time it updates thresholds. Additionally, we estimate that the number of fetched data from off-chip memory for randomForest and extraTrees is $KN \log L$ per tree because, in the best splitting case, the depth is $\log L$, and each node has K different candidate attributes (see Section 2). Also, randomForest and extraTrees execute random access because the subset of training data changes for each update, except for the root node.

The upper half of Table 9 gives the ratio of fetched training data from off-chip memory for each method when we set D to 15. RandomForest and extraTrees required 23.4 and 24.3 times more training data from off-chip memory than ExtraFerns, respectively. Therefore, ExtraFerns required only 4.3% and 4.1% off-chip memory accesses compared to randomForest and extraTrees, respectively.

3. Large N

This case corresponds to the original full datasets of HIGGS and SUSY. In this case, the number of fetched data from off-chip memory for ExtraFerns is UDN per tree because it is impossible to store DN samples in on-chip memory. Although ExtraFerns' threshold optimization requires U times more off-chip memory accesses than rFerns, the increase in accuracy far outweighs this disadvantage. In randomForest and extraTrees, each node must read data twice for node selection and data division because each node cannot store all data used in node selection due to too much data. Because the number of nodes is $O(N)$ from Table 2, we assume that the tree makes nodes up to max depth D, and the number of fetched data from off-chip memory is $2KND$ per tree.

Table 9: Ratio of Fetched Training Data from Off-Chip Memory

Case	Dataset	ExtraFerns	rFerns	randomForest	extraTrees
2	MNIST	**15**	**15**	333 (×22.2)	349 (×23.2)
	Fashion-MNIST	**15**	**15**	318 (×21.2)	339 (×22.6)
	Kuzushiji-MNIST	**15**	**15**	339 (×22.6)	347 (×23.1)
	Devanagari-Script	**15**	**15**	413 (×27.6)	425 (×28.3)
	Average	**15**	**15**	351 (×23.4)	365 (×24.3)
3	Original HIGGS	30	1	10	10
	Original SUSY	30	1	8	8
	Average	30	1	9	9

The lower half of Table 9 shows the ratio of training data amount fetched from off-chip memory for original HIGGS and SUSY. For comparison, we assume the same D for the four methods. The ratio of ExtraFerns is multiplied by U set to 30 in this paper. Both ratios of randomForest and extraTrees multiplied by $2K$, where we set 5 and 4 to K for HIGGS and SUSY, respectively. As a result, ExtraFerns requires 3 times and 3.75 times more data fetches for HIGGS and SUSY than randomForest and extraTrees need. Because randomForest and extraTrees require random access to off-chip memory, it is not always true that ExtraFerns is inferior to extraTrees or randomForest. Still, it is undeniable that ExtraFerns has a weak point against large-size datasets and needs to be improved.

5.4 Power Consumption

We can estimate the power consumption of ExtraFerns by multiplying the number of operations and the energy consumption for each operation. As mentioned in Section 2, however, the off-chip memory access operation takes 6400x more energy than the addition or the comparison operation needs. So even if we require 10-100x more computational operations than off-chip memory access, off-chip memory access is still the most dominant factor in power consumption. This case can happen only when the training data size from off-chip memory is small enough to store all of them in on-chip memory. So, in usual, the power consumption for computation is negligible in ExtraFerns, so that we can approximately estimate the total power consumption by considering only the off-chip memory access.

Figure 6 shows the estimated power consumption per tree for four datasets: MNIST, Fashion-MNIST, Kuzushiji-MNIST, and Devanagari-Script. From [9], we assume DRAM access consumes 20pJ/bit. In each graph, red bars and blue bars show the power consumption required to load training data from DRAM and write a trained model back to it. At a depth of 25, we excluded rFerns from evaluation targets because its model size is too large to store. In Figure 6, ExtraFerns shows the minimal power consumption for all four datasets thanks to the lightweight data read requirement. However, as mentioned in Section 5.2, the model size becomes more significant at a depth of 25. For two of the four datasets, the model size becomes larger than the amount of training data read, resulting in increased power consumption. The model size with large depths is a problem even in terms of power consumption.

6 Random Projection with ExtraFerns

As mentioned in Section 5, ExtraFerns achieves better memory access efficiency but still has room for improvement in memory usage. This section shows that random projection (RP) [6, 7] improves the accuracy of ExtraFerns with a low depth, especially on image datasets, resulting in memory usage reduction. This is not the ultimate solution for the problem but a piece of evidence that there still exist ways to enhance ExtraFerns.

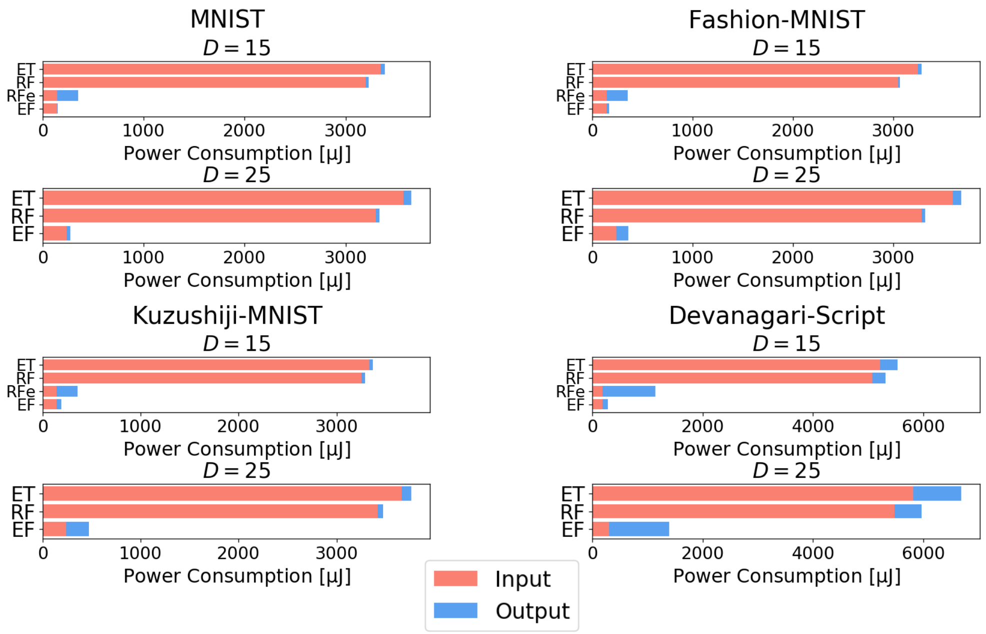

Figure 6: Power consumption comparison between MNIST, Fashion-MNIST, Kuzushiji-MNIST, Devanagari-Script. ET, RF, RFe, and EF represent extraTrees, randomForest, rFerns, and ExtraFerns. In each graph, red bars and blue bars show the power consumption required to load training data from DRAM and write a trained model back to it. From the result, we can confirm that ExtraFerns requires minimal power consumption in every condition.

6.1 Modified Random Projection for ExtraFerns

RP is a linear transform method usually used for dimensionality reduction [6, 7]. We propose applying a modified RP to ExtraFerns as a preprocessing step, which achieves higher accuracy. Figure 7 shows the difference between the original RP and our modified RP. As shown in Figure 7, given the original data matrix X, RP linearly transforms the original data X to the projection data X^{RP} with the random matrix R. The random matrix R is not necessarily square, but we use a square matrix $R_{k\times k}$ to preserve the original data's dimensionality. Also, sparse RP [7] stochastically defines non-zero r_{ij} as

$$r_{ij} = \begin{cases} +\sqrt{s} & \text{with probability } \frac{1}{2s} \\ 0 & \text{with probability } 1-\frac{1}{s} \\ -\sqrt{s} & \text{with probability } \frac{1}{2s} \end{cases}, \tag{9}$$

where the hyperparameter s is usually a large number proportional to the number of attributes. However, our RP leaves a constant number of non-zero r_{ij} in each row, where this constant is E in Figure 7. This simple modification allows better performance with ExtraFerns.

6.2 Effectiveness of Modified Random Projection

Figure 8 shows accuracy comparison results between bare ExtraFerns, ExtraFerns with the original RP, and ExtraFerns with the modified RP. The comparison is conducted on two configurations, $D = 15$ and $D = 25$, and six datasets: MNIST, Fashion-MNIST, Kuzushiji-MNIST, Devanagari-Script, HIGGS, and SUSY. In each comparison, we investigated accuracies with E in the range of 2 to 10. Also, we set s as half of the number of attributes for $E = 2$, one third for $E = 3$, etc. for fair comparison. Each row of Table 10 compares the ExtraFerns' accuracies with and without RP, where the accuracy with RP is the maximum value of each corresponding graph shown in Figure 8. The blue indicates the higher accuracy, and the red indicates the lower accuracy at the same depth.

As shown in Figure 8 and Table 10, the modified RP shows better performance than the original RP, improving the accuracy by up to 2.0% and 1.1% on average at $D = 15$ and $E = 2$, respectively for the four

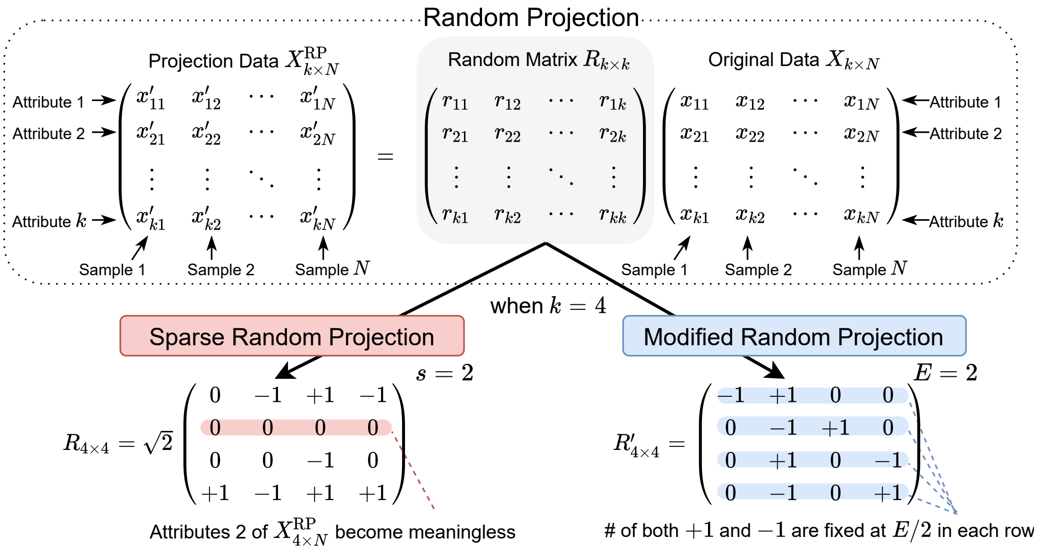
Random Projection
Projection Data $X^{\mathrm{RP}}_{k\times N}$
Random Matrix $R_{k\times k}$
Original Data $X_{k\times N}$
Attribute 1
Attribute 2
Attribute k
Sample 1
Sample 2
Sample N
=
when $k = 4$
Sparse Random Projection
Modified Random Projection
$s = 2$
$E = 2$
$R_{4\times4} = \sqrt{2}$
$R'_{4\times4} =$
Attributes 2 of $X^{\mathrm{RP}}_{4\times N}$ become meaningless
# of both $+1$ and -1 are fixed at $E/2$ in each row

Figure 7: Overview of RP for ExtraFerns. The k-dimensional original data with N samples $\boldsymbol{X}_{k\times N}$ are projected to $X^{\mathrm{RP}}_{k\times N}$ using a random $k \times k$-dimensional matrix $\boldsymbol{R}_{k\times k}$. The dimensionalities of the original data and the projection data are the same. The modified RP is a variation of sparse RP with a restriction on the number of non-zero values in each row vector and without the multiplication of $\sqrt{s}$ because quantization follows RP in ExtraFerns.

Table 10: Accuracy Comparison between ExtraFerns without RP and with RP

Dataset	Accuracy[%]			
	Depth 15		Depth 25	
	w/o RP	w/ RP	w/o RP	w/ RP
MNIST	93.9	95.9 (+2.0)	95.8	96.2 (+0.4)
Fashion-MNIST	82.5	83.7 (+1.2)	84.4	84.3 (-0.1)
Kuzushiji-MNIST	80.9	82.1 (+1.2)	84.2	83.8 (-0.4)
Devanagari-Script	88.9	89.0 (+0.1)	90.1	87.9 (-2.2)
HIGGS	68.0	63.3 (-4.7)	66.8	62.5 (-4.3)
SUSY	78.9	77.8 (-1.1)	79.1	78.5 (-0.6)

datasets: MNIST, Fashion-MNIST, Kuzushiji-MNIST, and Devanagari-Script. For the other two datasets, HIGGS and SUSY, both ExtraFerns with RP showed accuracy drops under all configurations, dropping by 2.9% on average at depth 15. The most significant difference between these two groups of datasets is their data type, i.e., the modified RP has good compatibility with the former four datasets because they contain image data with correlated attributes. However, when the depth is 25, RP is ineffective for three out of four image datasets, decreasing the accuracy by 0.9% on average. Therefore, RP is only effective for the image datasets when the depth of ExtraFerns is shallow. This means we can improve ExtraFerns' accuracy for image datasets by applying the modified RP instead of deepening the fern structure, saving significant memory usage.

6.3 ExtraFerns with Random Projection and RandomFerns

ExtraFerns with the modified RP includes randomFerns as a subset; when $E = 2$ and the threshold in each node is 0, the structure of ExtraFerns is identical to that of randomFerns. Therefore, we thought it would be an interesting analysis to compare ExtraFerns and randomFerns and investigate what happens if we merge the two methods. We implemented a program merging ExtraFerns and randomFerns in a certain ratio and evaluated it on image datasets. As mentioned above, randomFerns is not useful for tabular datasets.

Table 11 lists the accuracy comparison results at $D = 15$ on the four image datasets. In Table 11, the column

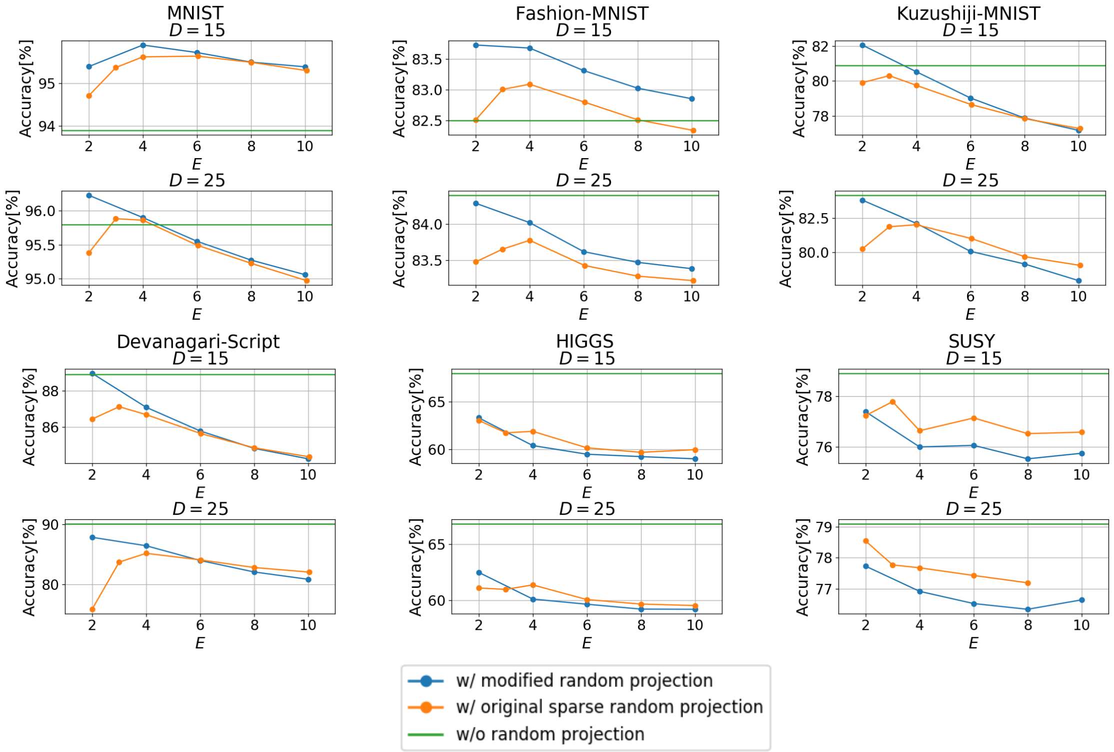

Figure 8: Experimental results for the effect of RP. The parameter E on the horizontal axes is the expected value of the number of non-zero values in each row of R. The RP was effective for datasets with large dimensions, and $D = 15$ was more effective than $D = 25$. In comparing the implementation of RP, the modified RP was more accurate for the datasets where RP was effective.

with a ratio of 0 represents bare ExtraFerns, and the column with a ratio of 1 represents bare randomFerns. The middle columns list the accuracies of the methods merged with the corresponding ratios. Also, the right-most column represents ExtraFerns with RP. From Table 11, we can confirm that ExtraFerns with RP achieves higher accuracy than other variations and is more effective than randomFerns, even on image datasets.

7 Conclusion

This paper proposed an edge-oriented decision tree ensemble called ExtraFerns that requires extremely low memory access for training. For achieving high inference accuracy, ExtraFerns conducts non-greedy optimization so that it can significantly outperform rFerns based on the fern structure. Experimental results show that ExtraFerns required only 4.3% and 4.1% memory access for training models only with 3.0% and 1.2% accuracy drops compared with randomForest and extraTrees, respectively. However, we also found that ExtraFerns has a weakness in memory usage, which increases significantly with its depth. To increase the accuracy without an increase in the depth, we proposed applying modified RP to ExtraFerns and showed that RP improves ExtraFerns' accuracy by up to 2.0% on the image datasets, which have correlated attributes. Our future work will aim to achieve further accuracy improvement and lower memory usage.

Acknowledgement

We are grateful to Prof. Ichigaku Takigawa for suggesting random projection with ExtraFerns. This work was partially supported by JST CREST Grant Number JPMJCR18K3, Japan.

Table 11: Accuracy Comparison between ExtraFerns with RP and ExtraFerns Combined with RandomFerns at Different Ratios

Dataset	Accuracy[%]					
	+RandomFerns' nodes					+Random projection
	Ratio of randomFerns' nodes					
	0	0.25	0.5	0.75	1	
MNIST	93.9	**94.0**	93.8	93.8	93.8	95.9
Fashion-MNIST	82.5	**82.6**	82.4	82.5	82.3	83.7
Kuzushiji-MNIST	**80.9**	80.7	80.3	79.8	79.3	82.1
Devanagari-Script	**88.9**	88.7	88.5	88.1	87.8	89.0

References

[1] Shungo Kumazawa, Kazushi Kawamura, Thiem Van Chu, Masato Motomura, and Jaehoon Yu. ExtraFerns: Fully parallel ensemble learning technique with non-greedy yet minimal memory access training. In *Proceedings of International Symposium on Computing and Networking*, pages 146–152, 2020.

[2] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001.

[3] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, March 2006.

[4] Mustafa Ozuysal, Pascal Fua, and Vincent Lepetit. Fast keypoint recognition in ten lines of code. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[5] Miron Bartosz Kursa. rFerns: An implementation of the random ferns method for general-purpose machine learning. *Journal of Statistical Software*, 61(10):1–13, November 2014.

[6] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 245–250, August 2001.

[7] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 287–296, August 2006.

[8] Gilles Louppe. Understanding random forests: From theory to practice. PhD thesis, University of Liège, 2014.

[9] Mark Horowitz. Computing's energy problem (and what we can do about it). In *Proceedings of IEEE International Solid-State Circuits Conference*, pages 10–14, 2014.

[10] Dheeru Dua and Casey Graff. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2017.

[11] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. arXiv:1708.07747, 2017.

[12] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep learning for classical japanese literature. arXiv:1812.01718, 2018.

[13] Shailesh Acharya, Ashok Kumar Pant, and Prashnna Kumar Gyawali. Deep learning based large scale handwritten Devanagari character recognition. In *Proceedings of the International Conference on Software, Knowledge Information, Industrial Management and Applications*, pages 1–6, 2015.

[14] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308, July 2014.