Journal of Parallel and Distributed Computing 132 (2019) 113–126

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

ForestLayer: Efficient training of deep forests on distributed task-parallel platforms

Guanghui Zhu, Qiu Hu, Rong Gu, Chunfeng Yuan, Yihua Huang [*]

National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

HIGHLIGHTS

- ForestLayer is an efficient and scalable system for the training of deep forests.
- Better performance than gcForest with 7x to 20.9x speedup on a range of datasets.
- Outperform tfForest and achieves better predictive performance.
- Achieve near linear scalability and good load balance.
- High ease-of-use with a set of high-level programming APIs.

ARTICLE INFO

Article history:
Received 31 July 2018
Received in revised form 15 April 2019
Accepted 1 May 2019
Available online 8 June 2019

Keywords:
Deep forest
Distributed computing
Task-parallel
Random forest
Ray

ABSTRACT

Most of the existing deep models are deep neural networks. Recently, the deep forest opens a door towards an alternative to deep neural networks for many tasks and has attracted more and more attention. At the same time, the deep forest model becomes widely used in many real-world applications. However, the existing deep forest system is inefficient and lacks scalability. In this paper, we present ForestLayer, which is an efficient and scalable deep forest system built on distributed task-parallel platforms. First, to improve the computing concurrency and reduce the communication overhead, we propose a fine-grained sub-forest based task-parallel algorithm. Next, we design a novel task splitting mechanism to reduce the training time without decreasing the accuracy of the original method. To further improve the performance of ForestLayer, we propose three system-level optimization techniques, including lazy scan, pre-pooling, and partial transmission. Besides the systematic optimization, we also propose a set of high-level programming APIs to improve the ease-of-use of ForestLayer. Finally, we have implemented ForestLayer on the distributed task-parallel platform Ray. The experimental results reveal that ForestLayer outperforms the existing deep forest system gcForest with 7× to 20.9× speedup on a range of datasets. In addition, ForestLayer outperforms TensorFlow-based implementation on most of the datasets, while achieving better predictive performance. Furthermore, ForestLayer achieves good scalability and load balance.

1. Introduction

Deep learning, one of the hottest research topics in the field of artificial intelligence, has achieved great success in many applications such as computer vision [20,23,29], speech recognition [21,35] and natural language processing [16,34]. It is well known that the core of deep learning is the deep neural networks (DNNs). Although DNNs are powerful in representation learning, they have apparent shortcomings [40]. The DNN is inapplicable to tasks with small-scale training data because the training of DNNs requires a large amount of training data. Furthermore, the powerful computational facilities such as GPU and TPU need to be used to train DNNs, which leads to huge economic costs. Most importantly, DNNs have too many hyper-parameters, and the tuning of hyper-parameters is a tricky process without theoretical guarantees.

Since the aforementioned problems of DNNs are intractable, many researchers began to explore alternatives to DNNs. Recently, Zhou and Feng proposed deep forest [40], which opens a door towards an alternative to DNNs for many tasks. Deep forest is an ensemble of random forests, with a cascade forest structure to perform representation learning. In contrast to DNNs, the model complexity of deep forest such as the number of

* Corresponding author.
E-mail addresses: guanghui.zhu@smail.nju.edu.cn (G. Zhu), huqiu@smail.nju.edu.cn (Q. Hu), gurong@nju.edu.cn (R. Gu), cfyuan@nju.edu.cn (C. Yuan), yhuang@nju.edu.cn (Y. Huang).

https://doi.org/10.1016/j.jpdc.2019.05.001
0743-7315/© 2019 Elsevier Inc. All rights reserved.

cascade levels can be adaptively determined. In addition, deep forest can achieve highly competitive performance to DNNs in a broad range of tasks. Moreover, deep forest has much fewer hyper-parameters than DNNs, and the performance of deep forest is quite robust to hyper-parameter settings.

Deep forest is a promising substitute for DNNs in many tasks. However, the existing deep forest system gcForest [18,40] is inefficient. First, the training process of deep forest in gcForest is sequential. In each cascade level, only one forest is trained at a time. Second, gcForest is a single-machine system which lacks scalability. For instance, gcForest takes near 10 hours to train the deep forest for the MNIST dataset using a commodity PC with 2 Intel Xeon 2.1 GHz CPUs (12 cores). For the CIFAR10 dataset, the training time can go up to more than one day. The inefficiency of gcForest may hinder the research and applications of deep forest to some extent. Meanwhile, more and more researchers have realized that the high-performance system plays an important role in the practical use of machine learning algorithms [30,31]. Recently, many parallel and distributed algorithms such as parallel random forest [12], parallel training of large-scale neural networks [10] and parallel matrix–matrix multiplication [13], have been proposed to improve the efficiency of machine learning. Therefore, an efficient deep forest system is also urgently needed.

In this paper, we present ForestLayer,[1] which is an efficient and scalable deep forest system built on distributed task-parallel platforms. ForestLayer splits each forest into sub-forests. The training of each sub-forest corresponds to a computing task. Then, all sub-forests are trained in parallel. The key contributions of our work are summarized as below:

- First, we propose a novel fine-grained sub-forest based task-parallel algorithm to improve the computing concurrency and reduce the communication overhead.
- Then, we propose a uniform task splitting mechanism to ensure the consistency of training results with the original method, while reducing the total training time cost.
- To further improve the performance of ForestLayer, we propose three system-level optimization techniques, including lazy scan, pre-pooling, and partial transmission.
- To improve the ease-of-use of ForestLayer, we propose a set of high-level programming APIs for the deep forest construction.
- Finally, we implement ForestLayer on the distributed task-parallel platform Ray and experimentally evaluate its performance. The experimental results reveal that ForestLayer outperforms the existing deep forest system gcForest with $7\times$ to $20.9\times$ speedup on a range of datasets. In addition, ForestLayer outperforms TensorFlow-based implementation on most of the datasets, while achieving better predictive performance. Furthermore, ForestLayer achieves near linear scalability and good load balance.

The rest of the paper is organized as follows. We introduce the preliminary of this research in Section 2. Section 3 presents the fine-grained sub-forest based task-parallel algorithm. The system design and optimization are elaborated in Section 4. The experimental evaluation results are presented in Section 5. Section 6 describes the related work. Finally, in Section 7, we conclude the paper and discuss the future work.

[1] ForestLayer is now available at https://github.com/PasaLab/forestlayer.

2. Preliminary

2.1. Deep forest

Deep forest is an ensemble of random forests, with a cascade structure which enables deep forest to perform representation learning [40]. Each level of the cascade is an ensemble of random forests. To satisfy the diversity of ensemble learning, each level is composed of different types of forests such as random forests [8] and completely-random forests [19]. For the classification tasks, each forest will output a class vector for an instance. Then, the class vector is concatenated with the original feature vector to be input to the next level of the cascade forest. Fig. 1 shows the overall procedure of deep forest. Suppose that there are three classes, then each of the four forests will produce a three-dimensional class vector. Each element of the class vector indicates the class probability. Thus, the next level of the cascade forest will receive 12 ($= 3 \times 4$) augmented features.

To enhance the representation learning ability for the inputs that have spatial and sequential relationships among raw features, a procedure of multi-grained scanning is adopted before the cascade forest. The multi-grained scanning stage is composed of two steps: window scanning and feature transformation. As Fig. 1 illustrates, three sliding windows are used to scan the raw features. Suppose that there are sequence data with 400 dimensions for every instance and a 100-dimensional sliding window is used. Then, for a raw input instance, 301 instances will be generated and the dimension of each instance is 100. The 301 instances are assigned with the same label as the raw 400-dimensional instance. The instances extracted from a sliding window will be used to train a random forest and a completely-random forest. Then, the class vectors are generated and concatenated as the transformed feature vectors. By using different sizes of sliding windows, deep forest will produce different grained feature vectors. To reduce the risk of overfitting, the class vectors corresponding to each forest are generated by k-fold cross-validation.

In deep forest, the model complexity (i.e., the number of cascade levels) is automatically determined. After expanding to a new cascade level, the performance of the whole cascade forest will be estimated on the validation set. If there is no significant performance gain during a few consecutive levels, the training process will be terminated.

As we can see above, the training process of deep forest is very computation-intensive. However, the existing deep forest system gcForest [18,40] is not very efficient. To improve the training efficiency of deep forest, in this paper, we propose a fine-grained task-parallel distributed deep forest system, ForestLayer.

2.2. Ray

Each cascade level of deep forest is composed of multiple independent random forests. Moreover, each forest usually contains hundreds of independent decision trees. Thus, the training of deep forest is well-suited for task-parallel computation.

Ray [25,27] is a high-performance distributed execution framework designed for emerging machine learning applications. In contrast to the data-parallel computing frameworks such as Apache Spark [37], Apache Hadoop [2], Ray naturally supports the task-parallel computing mode. In addition, Ray provides actor programming abstraction to support stateful components. In ForestLayer, the actor programming model can be used to wrap random forests with k-fold cross-validation. Thus, we choose Ray as the underlying distributed computing framework.

Ray has three significant properties for highly efficient distributed computing. First, Ray provides horizontal scalability with

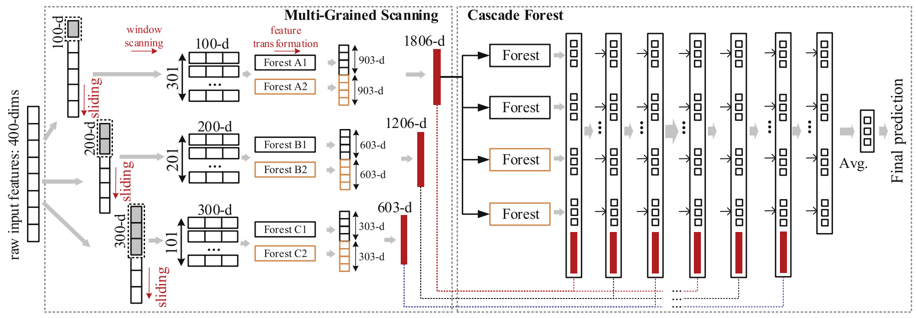


Fig. 1. Deep forest architecture.

a global control state store. Each computing node is stateless. As a result, the size of the distributed computing cluster can easily be scaled horizontally. Second, Ray supports low-latency and high-throughput task scheduling with a bottom-up distributed scheduler. The local scheduler schedules tasks locally or delegates tasks to the global scheduler according to the available computing resources. At last, Ray stores the inputs and outputs of every task with an in-memory distributed storage system. The object store is based on shared memory, which allows zero-copy data sharing between tasks running on the same node. Ray utilizes Apache Arrow [3] to implement the object store.

3. A fine-grained task-parallel algorithm

In this section, we present a fine-grained task-parallel algorithm in the distributed environment to train deep forest efficiently. The core of the task-parallel algorithm is the *split–merge* procedure. We first split each forest of the cascade level into many uniform sub-forests. Then, all sub-forest training tasks are performed in parallel. At last, the training results (i.e. class vectors) of each sub-forest are merged.

3.1. Motivation

In each cascade level of deep forest, all forests are independent. Thus, the training process of all forests is inherently parallel. Each forest corresponds to a computing task. We call this method the forest-based task-parallel algorithm (FTA, in short). However, in the distributed computing environment, FTA is not very efficient and scalable. In each cascade level, the degree of computation parallelism of FTA equals the total number of forests. In practice, the number of forests is very small. As a result, the cluster computing resources cannot be fully exploited due to the low degree of computation parallelism. For example, when the number of forests is less than the size of the cluster (i.e., the available computing nodes), many computing nodes will be idle.

In fact, the random forest is an ensemble of decision trees. Each tree can be trained in parallel. Thus, each tree of each forest represents a computing task. We call this method the tree-based task-parallel algorithm (TTA, in short). Suppose that each cascade level has F forests and T trees per forest. The degree of computation parallelism of TTA is $F * T$. Thus, TTA has a higher degree of parallelism and better resource utilization than FTA. However, the network communication overhead of TTA is significant due to the huge intermediate results produced by all trees. Moreover, a large number of computing tasks will lead to additional scheduling overhead.

To address the problem of FTA and TTA, in this paper, we propose a sub-forest based task-parallel algorithm (S-FTA). Each forest is split into multiple sub-forests. Each sub-forest corresponds to a computing task. The class vectors produced by each sub-forest are then merged. Next, we introduce the core *split–merge* procedure of S-FTA.

3.2. Split and merge

3.2.1. Split

To enhance the degree of computation parallelism and reduce the communication overhead simultaneously, we first split the random forest into sub-forests. There are two fundamental principles in the sub-forest splitting process.

- Principle 1 (P1). The class vectors produced by the sub-forest splitting method should be consistent with the not-splitting method.
- Principle 2 (P2). The sub-forest splitting method should minimize the expected training time.

To achieve these two principles, we design a sub-forest splitting mechanism. First, we propose a deterministic random state generation method for each sub-forest to ensure the consistency. Second, we propose a uniform splitting mechanism based on the weighted balls-into-bins solution to minimize the expected training time.

P1. In S-FTA, each sub-forest is trained independently. The training process of the sub-forest is similar to that of the whole forest. The input of each tree in the sub-forest is the bootstrap samples of the original input. For the sub-forest of the random forest, $\sqrt{K}$ features are randomly selected as candidates (K is the number of input features) and the one with the best *gini* value is chosen for the split. Similarly, for the sub-forest of the completely-random forest, the split feature at each node of the tree is randomly selected [40].

Note that the training results of each tree in a forest are determined by the initial random state of the forest. It is well known that a pseudo-random value is created by a deterministic algorithm in the computer system. Thus, to ensure that the splitting method and the non-splitting method have consistent training results, we elaborately set appropriate random states for each sub-forest.

Fig. 2 shows an example of S-FTA. Suppose that there is a forest f with 5 trees. Using the splitting method, we split the forest into sub-forests f_1, f_2, f_3 with 2, 2, 1 trees respectively. If the random state of f is s_0, then each tree of f is with the random state s_i ($1 \leq i \leq 5$), which is generated in the sequential order. We can explicitly set the initial random state of f_1 to s_0, the initial random state of f_2 to s_2, and the initial random state of f_3 to s_4. As a result, the random state of each tree after the sub-forest splitting is exactly the same as the non-splitting method. Thus, f_1, f_2, f_3 can be trained in parallel. The training result of each sub-forest will be merged into the final result, which is exactly the same as the result produced by f.

Moreover, each sub-forest needs to perform k-fold cross-validation to generate class vectors. Suppose that we conduct 3-fold cross-validation. We can set an *identical* group of initial random states for the cross-validation of sub-forests. Thus,

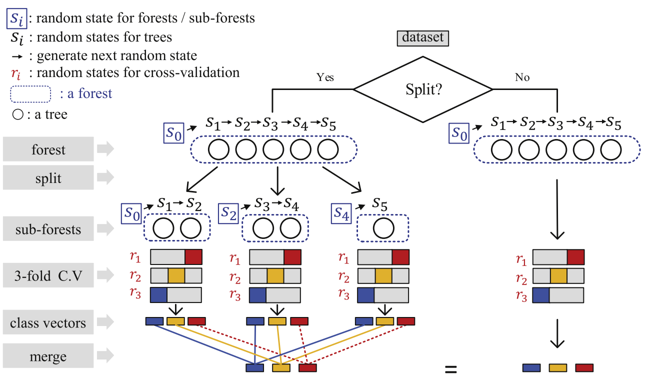

$\boxed{S_i}$: random state for forests / sub-forests
S_i : random states for trees
→ : generate next random state
r_i : random states for cross-validation
: a forest
○ : a tree
dataset
Split?
Yes
No
forest
split
sub-forests
3-fold C.V
class vectors
merge
$S_1 \to S_2 \to S_3 \to S_4 \to S_5$
S_0
$S_1 \to S_2$
$S_3 \to S_4$
S_5
S_2
S_4
r_1
r_2
r_3
=

Fig. 2. Consistency of the training results between the splitting method and the non-splitting method. The forest contains 5 trees and 3-fold cross-validation (C.V. for short) is used.

each sub-forest will produce exactly the *same* data for each cross-validation fold.

For instance, as Fig. 2 illustrates, we set the random state r_1, r_2, r_3 for each fold of sub-forests f_1, f_2, f_3 and non-split forest f. Thus, the final result is consistent with the non-splitting method by merging the results of each cross-validation fold.

P2. In ForestLayer, the underlying distributed task-parallel execution framework is Ray. Each sub-forest corresponds to an actor in Ray. The creation of an actor in Ray will be randomly scheduled to one node in the cluster [25]. Under this circumstance, we assume that all training tasks are randomly and independently allocated to computing nodes. Our goal is to minimize the expected maximum load of any node by splitting forests into multiple sub-forests, each of which contains a specific number of trees.

We model the aforementioned problem as the weighted balls-into-bins game [6,7]. A sub-forest corresponds to a ball and the computing nodes correspond to bins. Since sub-forests of different sizes have different resource requirements (i.e., memory and CPUs), we view each sub-forest as a weighted ball, where the weight can be represented as the size of the sub-forest.

We formally describe the problem of the weighted balls-into-bins game. Suppose that we have m balls and n bins. Ball b has weight w_b for all $b \in \{1, \dots, m\}$. Let $w = (w_1, \dots, w_m)$ be the weight vector of balls, $W = \sum_{i=1}^{m} w_i$ is the total weight of all balls. The load of a bin is the sum of the weights of all balls allocated to it. The status of an allocation is represented by a load vector $L(w) = (l_1(w), \dots, l_n(w))$, where $l_t(w)$ is the load of the tth bin after the allocation of the weight vector w. We normalize the load vector L with a non-increasing order of bin loads, i.e., $l_1 \geq l_2 \geq \cdots \geq l_n$. Moreover, we denote $S_i(w) = \sum_{j=1}^{i} l_j(w)$ as the total load of the i highest-loaded bins.

Theorem 1. *If $w \succ w'$, then $E[S_i(w')] \leq E[S_i(w)]$ for $\forall i \in \{1, \dots, n\}$ [6].*

Theorem 1 has been proved by Petra and Tom [6]. In Theorem 1, $w \succ w'$ means that w majorises w'. If w majorises w', we can say that the weight vector w' is more balanced than w. It is clearly that a *uniform* weight vector is majorised by all other weight vectors. Thus, from Theorem 1 we can conclude that *uniform* balls can minimize the expected maximum load. Similarly, for the sub-forest splitting problem, the *uniform* sub-forests with the same number of trees can minimize the expected maximum load and thus can minimize the expected training time.

Inspired by the above theorem, we split forests into *uniform* sub-forests to minimize the expected training time. Let g denote the split granularity, which indicates the number of trees in each sub-forest. For instance, suppose that the split granularity g is 250. Then, a forest of 500 trees will be split into 2 sub-forests, each of which contains 250 trees.

3.2.2. Merge

After the sub-forest splitting, the intermediate results produced by each sub-forest need to be merged. For a sub-forest with t trees, the output of an instance is $o(t) = \frac{1}{t} \sum_{i=1}^{t} \boldsymbol{c}_i$, where $\boldsymbol{c}_i$ is the class vector produced by the ith tree in the sub-forest. If the original forest is with T trees and is split into K sub-forests. After the merging process, the final class vector is as follows:

$$\boldsymbol{c} = \sum_{k=1}^{K} \frac{t_k}{T} * o(t_k) = \frac{1}{T} \sum_{k=1}^{K} \sum_{i=1}^{t_k} \boldsymbol{c_i}$$

where t_k denotes the number of trees in the kth sub-forest ($\sum_{k=1}^{K} t_k = T$).

A naive merging strategy is that the computing workers directly send results to the master, and then the master merges all of the intermediate results. However, when the size of the intermediate results goes huge, the master may become a performance bottleneck. To address the potential bottleneck, we employ the tree-reduce aggregation approach [9,25,37], in which the results are aggregated in a hierarchical bottom-up tree mode. Specifically, we encapsulate the merge operation as a computing task. The results of sub-forests are first sent to the bottom workers. The local merged results are further sent to the upper workers. At last, the master node merges all of the locally merged results. Fig. 3 shows the data flow of the *split–merge* procedure. There are two forests, each of which is split into 4 sub-forests. The results (A1–A4, B1–B4) are the training results of the sub-forests. The training results are merged with the tree-reduce approach and thus the workload of the master are greatly reduced.

3.3. Algorithm analysis

First, we discuss the degree of computation parallelism of S-FTA. Let C denote the number of classes, N the total number of training instances. Note that the dimension of class vectors produced by each forest equals C. Moreover, let F be the number of forests in each cascade level, T_i the size of ith forests, g the split granularity. Each sub-forest has g trees. Thus, the degree of computation parallelism is $\sum_{i=1}^{F} \lceil \frac{T_i}{g} \rceil$, which is between F and $\sum_{i=1}^{F} T_i$.

Next, we present the communication complexity of S-FTA. In deep forest, each cascade level produces F class vectors for an instance. For the FTA, there is no merging process. Thus, the communication complexity of FTA is $O(NCF)$. For the TTA, the class vectors produced by all trees need to be merged. The corresponding communication complexity of TTA is $O(NC \sum_{i=1}^{F} T_i)$. Compared to FTA and TTA, the communication complexity of S-FTA is $O(NC \sum_{i=1}^{F} \lceil \frac{T_i}{g} \rceil)$. Specifically, each sub-forest produces an intermediate class vector (i.e., the sum of class vectors produced by each tree in the sub-forest) for each input instance. All the intermediate class vectors are further merged and averaged. The final class vector is the averaged class vector.

From the above analysis, we can see that the split granularity g plays an important role in S-FTA. There is a trade-off between the degree of computation parallelism and the communication overhead. When the split granularity decreases, the degree of parallelism becomes larger and thus the load in each computing node is more balanced. However, more communication overhead will be caused due to the smaller split granularity. The optimal split granularity g is related to the characteristics of datasets (i.e., the size and the dimension) and the size of the computing cluster. We empirically analyze the split granularity in Section 5.

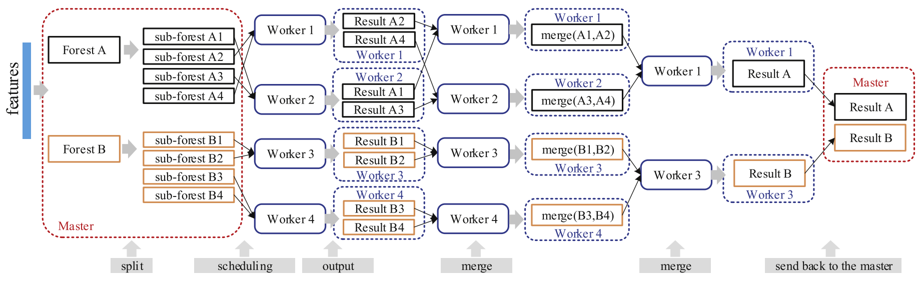


Fig. 3. *split–merge* dataflow. There are two forests, each of which is split into 4 sub-forests.

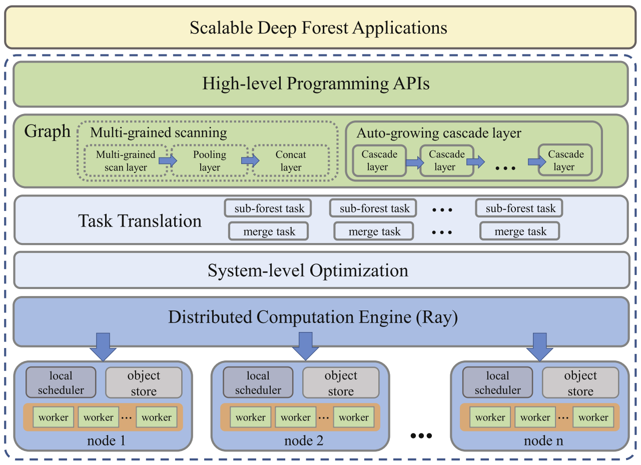


Fig. 4. System architecture.

4. System design and optimization

In this section, we introduce the system design and optimization of ForestLayer. First, we propose a set of high-level programming APIs for easy and fast construction of deep forests. To further improve the efficiency and scalability of ForestLayer, we propose three system-level optimization techniques for training deep forest in the distributed environment.

4.1. System overview

We have implemented ForestLayer on the top of the distributed execution framework Ray. Fig. 4 illustrates the overview of the system architecture, which consists of high-level programming APIs, task translation, system-level optimization, and distributed execution. Users can construct the deep forest using high-level programming APIs easily.

Specifically, the deep forest is represented as a computation graph that is a linear stack of layers. The computation graph will be translated into a series of computing tasks with the *split–merge* procedure (see Section 3). The computing tasks are composed of sub-forest training tasks and result merge tasks. After the system-level optimization, the training of the deep forest is performed on the distributed framework Ray.

4.2. High-level programming API

In this subsection, we first introduce the high-level programming APIs of ForestLayer. Then we introduce the underlying API differences for deep forests between the Ray-based design and the TensorFlow-based design. Then we introduce the Ray-based API design for deep forests and compare it with the TensorFlow-based API design.

Table 1
An overview of high-level programming APIs of ForestLayer.

Categories	APIs

Layer creation	`mgs = MultiGrainScanLayer(...)`
	`pool = PoolingLayer(...)`
	`concat = ConcatLayer(...)`
	`cas = AutoGrowingCascadeLayer(...)`

Graph creation	`g = Graph(pre_pool=True, . . . )`
	`g.add(<Layer>)`

Env. Initialization	`forestlayer.init(redis_address=...)`

Training	`<Graph|Layer>.fit(X, y)`
	`<Graph|Layer>.fit_transform(X, y, ..)`

Evaluation	`<Graph|Layer>.evaluate(X, y)`
Dataset loading	`uci_adult.load_data()`

4.2.1. High-level programming APIs of ForestLayer

The deep forest consists of two parts: multi-grained scanning and the cascade forest. In ForestLayer, we first propose the layer abstraction for the two parts respectively. Specifically, the multi-grained scanning contains three types of layers: *MultiGrainScanLayer*, *PoolingLayer*, and *ConcatLayer*, where *MultiGrainScanLayer* scans the raw input with different sliding windows, *PoolingLayer* subsamples the instances generated by the window scanning, and *ConcatLayer* concatenates the class vectors produced by each forest. The results of the multi-grained scanning are the input of the cascade forest. Each level of the cascade forest is an ensemble of random forests. We represent each level as *CascadeLayer* and represent the whole cascade forest as *AutoGrowingCascadeLayer*. Thus, the *AutoGrowingCascadeLayer* is a linear stack of *CascadeLayer*.

We use the computation graph to encapsulate the aforementioned layers. For instance, given the sequence data or the image data, the corresponding computation graph is composed of *MultiGrainScanLayer* and *AutoGrowingCascadeLayer*. In addition, the shared parameters and the global states are stored in the computation graph.

Besides the layer and graph abstraction, to support the end-to-end deep forest application, we also provide a set of high-level APIs for data loading, model training, and model evaluation. Table 1 shows an overview of high-level APIs provided by ForestLayer. The high-level programming APIs enable users to construct deep forests easily and allow for fast experimentation with deep forests.

4.2.2. Ray-based and TensorFlow-based API design for deep forest

The core of the deep forest architecture is the cascade forest. In contrast to most deep neural networks whose model architecture is fixed, the number of cascade layers in the cascade forest can be adaptively determined [40]. In this subsection, we focus on the cascade forest and describe how to construct a cascade layer using Ray-based and TensorFlow-based APIs.

Fig. 5 shows the comparison between Ray-based and TensorFlow-based API design. For the Ray-based API design,

```
# Initialize the distributed environment of Ray
ray.init(redis_server=..)

class CascadeLayer():
  def __init__(self, forests):
    # Initialize the cascade layer.
    self.forests = forests
  def fit_transform(self, X, y):
    # Each sub-forest is a remote Actor
    sub_forests = split(self.forests, granularity)
    # Put raw input data into shared memory store
    X_id, y_id = ray.put(X, y)
    # Execute remote function in parallel
    outputs = [sf.fit_transform.remote(X_id, y_id)
                 for sf in sub_forests]
    # Merge the class vectors
    merge_result_ids = merge.remote(outputs)
    # Return the final results in a blocking manner
    return ray.get(merged_result_ids)

@ray.remote
def merge(class_vectors):
  # Merge class vectors generated by sub-forests
  return merged_class_vector

@ray.remote
class Subforest(object):
  def __init__(self):
    # Initialize the sub-forest.
  def fit_transform(self,X, y):
    # Generate class vector with k-fold C.V.
    return class_vector
```

Ray API

```
# Configure distributed environment manually
cluster = tf.train.ClusterSpec({
  "ps": ps1:port,ps2:port,...,
  "worker": w1:port,w2:port,w3:port,...,})

server = tf.train.Server(cluster, job_name, task_idx)
sess = tf.Session(target=server.target)

if FLAGS.job_name =="ps":
  with tf.device("/job:ps"):
    # Store generated class vectors by all workers
    class_vectors = [tf.Variable(..), ..]
    # Indicates whether a worker completes work
    ready = [tf.Variable(False), ..]
  sess.run(tf.global_variables_initializer())
  # Waiting for all workers finished
  while not sess.run(ready).all(): sleep(1)
  # Aggregate and merge class vectors of this layer
elif FLAGS.job_name =="worker":
  # Load training data X, y
  class_vector = Subforest(..).fit_transform(X, y)
  # Get the slot that this worker need to update
  x = get_variable("x_{task_index}", reuse=True)
  sess.run(x.assign(class_vector))
  sess.run(ready[task_index].assign(True))

class Subforest(object):
  def __init__(self):
    # Initialize the sub-forest
  def fit_transform(self, X, y):
    # Generate class vector with k-fold CV
    return class_vector
```

TensorFlow API

Fig. 5. Comparison between Ray-based and TensorFlow-based APIs design for deep forests.

`@ray.remote` indicates remote functions and actors. Invocations of remote functions and actor methods return futures, which can be passed to subsequent remote functions or actor methods to encode task dependencies. We can use the `ray.get` API to get the values of futures. Each remote function or actor method corresponds to a computing task, which can be executed in a distributed fashion. We implement the sub-forest as a remote actor (`Subforest`) and the sub-forest training task (`Subforest::fit_transform`) as a remote actor method. The sub-forest training tasks will be dispatched dynamically to cluster nodes by a distributed scheduler. The computation results (i.e., class vectors) of sub-forest training tasks are managed by the in-memory distributed object store of Ray. In addition, we also implement the tree-reduce merge process as a remote function (`merge`).

For the TensorFlow-based API design, we also use the same *split–merge* procedure as proposed in ForestLayer. In TensorFlow, we should first configure the cluster specifications manually, which can be tedious, especially for large clusters. Additionally, without a flexible distributed scheduler, we need to manually assign each sub-forest training task to a specific worker, which imposes additional burdens on developers. We use the built-in parameter server (`ps`) to manage the class vectors produced by each sub-forest. The `ps` tasks and the sub-forest training tasks are distinguished by the input job name. Moreover, different sub-forest training tasks have different task ids.

In summary, the overall comparison between Ray-based and TensorFlow-based APIs for deep forests is shown in Table 2. We compare the Ray-based APIs with the TensorFlow-based APIs in terms of programming flexibility, task-parallel computation, distributed training, task scheduling, and shared data management. From Table 2, we can see that the programming flexibility of Ray is high. Moreover, Ray is more suitable for distributed task-parallel computation than TensorFlow. Therefore, we choose Ray as the distributed framework for constructing deep forests.

4.3. System-level optimization

We propose three system-level optimizations to reduce the network communication overhead in the distributed computing environment.

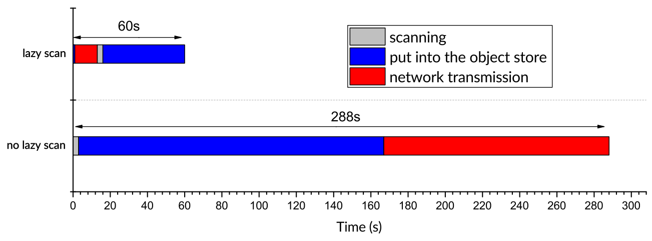


Fig. 6. Comparison of the preprocessing time at the multi-grained window scanning stage. Dataset: CIFAR10.

4.3.1. Lazy scan

To deal with the inputs that have spatial and sequential relationships between raw features, we first use the multi-grained scanning to preprocess the raw input data [40]. However, the total amount of data after the multi-grained window scanning are usually dozens or hundreds of times of the original input data. Thus, transferring the huge scanning results to each computing worker will cause significant network communication overhead.

To tackle this problem, we present a lazy scan scheme to postpone the multi-grained scanning. Since the cost of the window scanning is relatively cheap, it is unnecessary to perform the multi-grained window scanning in advance. We can perform the window scanning concurrently in all computing workers when the scanning results are needed. Thus, the transmission of the huge scanning results can be avoided and we only need to transfer the original input data once. With the lazy scan optimization, the communication cost can be significantly reduced.

Furthermore, to avoid the unnecessary memory overhead, we keep exactly one copy for the scanning result of each sliding window in a computing node. We put this copy into the object store, which is the in-memory distributed storage system of the Ray framework. The object store is based on the shared memory. Thus, the scanning results can be shared between every worker process in the computing node.

Fig. 6 describes the comparison of the preprocessing time between lazy scan method and the original scan method. The aim of the preprocessing is to scan the input instance with multi-grained sliding windows. The scanning results are used to train the subsequent random forests. From Fig. 6, we can see that the

Table 2
Overall comparison between Ray-based and TensorFlow-based API for deep forests. The task indicates the sub-forest training task.

	Programming flexibility	Task-parallel computation	Distributed training	Task scheduling	Shared data management
Ray	High	Natively supported	Supported	Automatic scheduling with distributed task scheduler	In-memory distributed object store
TensorFlow	Low	Supported	Supported	Manual scheduling	Parameter server

lazy scan method outperforms the original scan method with 4.8× speedup. The raw input data are scanned locally only when the multi-grained scanning is needed. Thus, compared to the original scan method, the lazy scan method takes less time in both the object store step and the data transmission step.

4.3.2. *Pre-pooling*

After the sliding window scanning, the instances will be used to train a random forest and a completely-random forest. Then, the class vectors are generated and concatenated as transformed features [40]. Note that the dimension of transformed feature vectors is much higher than the raw features. In some cases, the transformed feature vectors are too long to be accommodated. Thus, we need to perform feature pooling (i.e., subsampling) to reduce the dimension of transformed features. In order to reduce the communication overhead during the process of class vector concatenation, we propose a pre-pooling optimization technique.

Since we employ a declarative programming paradigm (see Section 4.2) to construct deep forests, whether to perform feature pooling is known in advance. As discussed in Section 3, each forest is split into multiple sub-forests. Therefore, if the feature pooling is required, we can perform pooling on the intermediate results produced by each sub-forest in advance. After the pre-pooling, the size of the intermediate results will become much smaller. Compared to the method that first merges intermediate results and then performs feature pooling, the pre-pooling method can reduce the communication overhead significantly.

For instance, the input is a 20×20 image. A 10×10 sliding window with stride 1 will produce 11×11 feature vectors. Suppose that there are three classes, each forest produces transformed class vectors with $11 \times 11 \times 3$ dimensions. If we perform a 2×2 non-overlapping pooling for each sub-forest with stride 2, the dimension of the transformed feature vector will become $6 \times 6 \times 3$. Thus, the total amount of data transmission can be reduced by 1/4 in the following class vector concatenation step.

4.3.3. *Partial transmission*

The partial transmission technique is effective in the cascade forest stage. The cascade forest stage consists of multiple cascade levels. In each cascade level, the class vectors produced by all forests are concatenated with the raw feature vectors. Then, the concatenated feature vectors will be fed into the next level of the cascade forest. Thus, we need to transfer the concatenated vectors to all computing nodes for the training of the next cascade level.

The concatenated vectors are composed of two parts: the class vectors and the raw feature vectors. In fact, the raw feature vectors are fixed. Each computing node can keep a copy of the raw feature vectors. However, the class vectors are dynamically produced in each cascade level. When training a new cascade level, we only transmit the dynamically updated class vectors to each computing node. It is unnecessary to transmit the raw feature vectors to each computing node repeatedly. Thus, we propose the partial transmission optimization, which only transmits the dynamically updated class vectors. In this way, the network communication cost can be reduced significantly. Fig. 10(b) experimentally demonstrates that the partial transmission technique is effective for performance improvement.

Specifically, the raw feature vectors are transmitted to all computing nodes only once before the training of the first cascade level. Each computing node keeps a copy of the raw feature vectors that are shared by all workers. Then, after each cascade level, only the latest class vectors are delivered to each worker. When workers receive the class vectors, they concatenate the class vectors with the local raw feature vectors to generate the input of the next cascade level.

Table 3
Dataset description.

Dataset	#train	#test	#features	#class
ADULT	32561	16281	14	2
YEAST	1038	446	8	10
LETTER	16000	4000	16	26
IMDB	25000	25000	5000	2
sEMG	1260	540	1×3000	6
MNIST	60000	10000	$28 \times 28 \times 1$	10
CIFAR10	50000	10000	$32 \times 32 \times 3$	10

5. Evaluation

In this section, we evaluate the performance of ForestLayer. For comparison, we also evaluate the performance of the existing deep forest system, gcForest. Moreover, we also implement the deep forest on TensorFlow and compare it with ForestLayer in the cluster of 16 nodes. For simplicity, we call the TensorFlow-based implementation tfForest. The highlights are summarized as follows:

1 ForestLayer outperforms gcForest with 7× to 20.9× speedup on a wide range of datasets (Section 5.3.1).

2 ForestLayer outperforms tfForest on most of the datasets, while achieving better predictive performance (Section 5.3.2).

3 ForestLayer achieves near-linear scalability and good load balance in the distributed computing environment (Section 5.4).

4 The proposed system-level optimization techniques are effective to improve the performance of ForestLayer (Section 5.5).

5 The effect of split granularity g on the overall performance is analyzed and discussed in detail (Section 5.6).

5.1. *Experiment setup*

All experiments are conducted on a physical cluster with 16 nodes. Each node has two Intel Xeon 2.1 GHz CPUs with 12 physical cores in total, 64 GB RAM, and a 6 TB RAID0 hard disk. All nodes are connected via 1 Gbps Ethernet. All the nodes run on the Ext4 file system and the RedHat7 operating system. The version of Ray is 0.4.0, gcForest is 1.0 [40] and TensorFlow is 1.12.0 [1].

We evaluate ForestLayer on a wide range of datasets. Since the prediction performance of deep forest has been evaluated in the prior work [40], we focus on the efficiency and scalability of ForestLayer. We use seven datasets in total. Table 3 lists the characteristics of each dataset, which includes the number of training examples, testing examples, features, and classes.

Table 4
Performance of ForestLayer and gcForest.

Dataset	Time unit	gcForest	ForestLayer					Speedup	Accuracy	#cascade levels
#nodes		1	1	2	4	8	16			
ADULT	sec	790.5	403.8	221.7	122.4	78.84	57.7	**13.7**×	86.0758%	5
YEAST	sec	288.9	60.5	37.7	23.7	16	13.8	**20.9**×	62.7803%	1
LETTER	sec	735.8	326.2	183.8	113	74.8	59.6	**12.3**×	97.45%	5
IMDB	min	144.7	132.7	76.6	36.53	22.25	14.1	**10.26**×	89.1320%	15
sEMG	min	600.2	589.2	587.8	458.5	144.3	85.8	**7.0**×	71.4815%	2
MNIST	min	566.7	547	465.4	253.3	119.6	64.4	**8.8**×	99.33%	18
CIFAR10	min	1526.2	1453.8	1209.9	759.1	324	164.9	**9.25**×	62.12%	14

Table 5
Performance of multi-grained scanning and cascade forest in minutes.

Dataset	gcForest		ForestLayer		MGS	CF	Overall
	MGS	CF	MGS	CF	speedup	speedup	speedup
MNIST	355.1	211.54	44.8	19.6	7.93×	10.79×	**8.8**×
CIFAR10	876	650.18	98.29	66.68	8.91×	9.75×	**9.25**×

We divide these datasets into two types. The first type is the tabular datasets, which include the UCI datasets [4] (i.e., ADULT, LETTER, and YEAST) and the IMDB dataset. The UCI datasets are low-dimensional and contain a relatively small number of features. The IMDB dataset is high-dimensional, which is used for sentiment analysis. The IMDB dataset contains 25000 movie reviews for training and 25000 movie reviews for testing. Each movie review in IMDB is represented by 5000 TF-IDF features.

The second type is the cognitive datasets, which include sEMG, MNIST, and CIFAR10. These datasets hold spatial or sequential relationships between the raw input features. sEMG [28] is a hand movement recognition dataset, where each instance contains a 3000-dimensional feature sequence. MNIST [24] and CIFAR10 [22] are image classification datasets with 10 classes. Each instance of MNIST is an image with the size of 28 × 28. Each instance of CIFAR10 is a colorful 32 × 32 image and each image has 3 channels (RGB).

5.2. Model configuration

We evaluate ForestLayer on two types of deep forest models. For the first type of datasets, only the cascade forest is used. For the second type of datasets which have spatial and sequential relationships among the raw input features, the multi-grained scanning is adopted before the cascade forest. Specifically, for the UCI datasets and the IMDB dataset, the model only consists of an auto-growing cascade layer. For the spatial datasets (i.e., MNIST, CIFAR10), and sequential datasets (i.e., sEMG), the deep forest is composed of a multi-grained scan layer, a pooling layer, a concat layer and an auto-growing cascade layer.

In the multi-grained scan layer, three sliding windows are used. For d raw features, we use sliding windows with size of $\lfloor d/16 \rfloor$, $\lfloor d/8 \rfloor$, $\lfloor d/4 \rfloor$. For example, for an image with size of 28×28 pixels, there are $\sqrt{\lfloor 28 \times 28/16 \rfloor} = 7$, $\sqrt{\lfloor 28 \times 28/8 \rfloor} = 9.9$, $\sqrt{\lfloor 28 \times 28/4 \rfloor} = 13$, the size of sliding windows are 7×7, 10×10, and 13×13. Followed by every window is a random forest and a completely-random forest. After the training of the two random forests, a pooling layer and a concat layer will be added. In the auto-growing cascade layer, each level has 8 forests (4 random forests and 4 completely-random forests). Each forest contains 500 trees. Three-fold cross validation is used for class vector generation.

For gcForest and tfForest, we use the same model configuration as ForestLayer. Particularly, in tfForest, the forest in both the multi-grained scanning layer and the auto-growing cascade layer is TensorForest [17,32], which an implementation of completely-random forests on TensorFlow.

5.3. Performance comparison

We evaluate the performance of ForestLayer on seven datasets and compare it with gcForest and tfForest.[2]

5.3.1. Comparison with gcForest

Table 4 shows the performance of ForestLayer and gcForest. ForestLayer outperforms gcForest with 7× to 20.9× speedup in the cluster of 16 nodes. Meanwhile, ForestLayer achieves exactly the same accuracy as gcForest with the same deep forest structure (i.e., the number of cascade levels).

From Table 4, on the one hand, we can see that ForestLayer outperforms gcForest even in the single-machine mode (i.e., only one computing node is used). The main reason is that the training process of deep forest in gcForest is sequential. In each cascade level, all forests are trained one by one. In contrast, ForestLayer first splits each forest into sub-forests and then trains all the sub-forests concurrently. Therefore, ForestLayer is more efficient than gcForest. Furthermore, as the number of computing nodes increases, the performance improvement becomes more significant.

On the other hand, for datasets such as ADULT, LETTER, YEAST, and IMDB, which do not have spatial or sequential relationships between the raw input features, ForestLayer runs an order of magnitude faster than gcForest. In contrast, for the image and sequence datasets that hold spatial or sequential relationships, the speedup is relatively small. The main reason is that the multi-grained scanning is needed for the image and sequence datasets. Table 5 illustrates the detailed performance of deep forests for the image datasets. Typically, the deep forest contains two stages: the multi-grained scanning stage (MGS stage for short) and the cascade forest stage (CF stage for short). In the cascade forest, ForestLayer can achieve an order of magnitude speedup over gcForest. However, the speedup of the MGS stage is not very significant, which hinders the overall performance improvement.

The main reason that the speedup of the MGS stage is not very significant is two-fold. On the one hand, the total amount of data (i.e., the number of training instances) produced by window scanning is hundreds of times larger than that of the original input instances. For instance, if a 400-dimensional original input instance is scanned by a 100-dimensional sliding window, then 301 instances will be produced after the window scanning. The scanning results using a specific window are then fed into random forests for the feature transformation. All forest training

[2] tfForest is now available at https://github.com/PasaLab/forestlayer/tree/master/tfForest.

tasks allocated in a node may correspond to different scanning windows. When these training tasks are performed in parallel, the computing node needs to store the scanning results of all scanning windows, which leads to huge memory overhead. Thus, we need to limit the degree of computation parallelism during the MGS stage. In the CF stage, there is no need to limit the degree of computation parallelism. This is because all training tasks in one node can share the same copy of input data. We can allocate more sub-forest training tasks to each node using a smaller split granularity. Thus, the CF stage has a high degree of computation parallelism.

On the other hand, due to the window scanning, the total number of training instances is very large. Correspondingly, the total number of class vectors produced by each sub-forest will also become large. Thus, huge communication overhead will be caused at the class vector merging process.

Overall, because of the low degree of computation parallelism and high communication overhead, the speedup of the MGS stage is relatively low. Therefore, compared to the other datasets that do not need the MGS stage, the performance improvement for the MNIST and CIFAR10 datasets is not very significant.

5.3.2. Comparison with tfForest

We also evaluate the performance of tfForest in the cluster of 16 nodes and compare it with ForestLayer. tfForest is built on distributed TensorFlow and uses TensorForest [17] as the basic random forest model. TensorForest is an implementation of random forests on TensorFlow, using an online, extremely randomized trees training algorithm. Table 6 shows the performance comparison between ForestLayer and tfForest. In terms of runtime, ForestLayer outperforms tfForest on most of the datasets. Moreover, ForestLayer achieves better prediction performance on all datasets.

From Table 6, we can see that ForestLayer runs faster than tfForest on 5/7 datasets except for the MNIST and CIFAR10 datasets. For the first type of datasets such as ADULT, LETTER, YEAST, and IMDB, which do not have spatial or sequential relationships between the raw input features, ForestLayer can achieve an order of magnitude speedup over tfForest. This is because these datasets are relatively small. For the small datasets, due to the larger system overhead from the TensorFlow platform, TensorForest runs slower than the random forest implementation in scikit-learn [17].

Next, we consider the sequence dataset (sEMG) and the image datasets (MNIST and CIFAR10) that hold sequential or spatial relationships. The deep forest architecture of these dataset consists of two stages: the MGS stage and the CF stage. Table 7 shows the runtime of each stage. We can see that tfForest achieves better runtime performance than ForestLayer in the MGS stage. Due to the window scanning, the size of input data for each forest in the MGS stage becomes very large. For the large datasets, TensorForest is more effective [17]. As a result, tfForest outperforms ForestLayer in the MGS stage. In the following CF stage, ForestLayer performs better than tfForest. But, the speedup is not significant except for the sEMG dataset. The sEMG dataset is a time-series dataset. After the feature transformation in the MGS stage, the input of the forest in the CF stage becomes high-dimensional. For instance, the dimension can be up to 16884. Since TensorForest utilizes an online, mini-batch training algorithm, it is not suitable for high-dimensional datasets [17]. Thus, for the sEMG dataset, ForestLayer achieves a 499.7× speedup in the CF stage and thus has better overall performance. Moreover, for the larger datasets MNIST and CIFAR10, tfForest achieves comparable performance with ForestLayer in the CF stage. But, due to much better performance in the MGS stage, tfForest is superior to ForestLayer in terms of overall runtime performance.

On the other hand, from Table 6, we can see that ForestLayer achieves better predictive performance (i.e., accuracy) on all datasets. The main reason is that TensorForest sacrifices a small percentage for its increased scalability [17]. TensorForest is more effective for very large datasets with fewer features at the cost of predictive performance.

5.4. Scalability and load balance

Scalability. We evaluate that node scalability of ForestLayer by scaling up the number of nodes over 2 types of datasets. Note that the second type of datasets (i.e., the sequence and image data) has spatial or sequential relationships between the raw input features. Fig. 7 demonstrates the node scalability of ForestLayer. The dashed line represents the ideal linear scalability.

The node scalability on the first type of datasets is shown in Figs. 7(a) and 7(b). It can be seen that ForestLayer scales near linearly when increasing the number of nodes. The node scalability on the second type of datasets is shown in Figs. 7(c) and 7(d). It can be seen that the performance improvement is not very significant on two nodes.

For the datasets (ADULT in Fig. 7(a) and IMDB in Fig. 7(b)) that do not have spatial or sequential relationships between raw features, the deep forest architecture only contains the cascade forest. In contrast, for the sequence dataset (sEMG in Fig. 7(c)) and the image dataset (CIFAR10 in Fig. 7(d)), the deep forest architecture consists of two stages: the MGS stage and the CF stage. Besides the overall node scalability, we also plot the node scalability of each stage in Figs. 8(a) and 8(b). We can see that the node scalability of the MGS stage is not linear especially on two nodes, which leads to the anomalies in the overall node scalability.

For the sEMG dataset, the overall performance on two nodes is almost the same as that on one node. The main reason is that the total number of cascade levels is small, the computation overhead of the MGS stage accounts for a large proportion of the total runtime. For the CIFAR10 dataset, the performance improvement is more significant on two nodes. This is because the number of cascade levels in the deep forest is relatively large and the performance improvement during the CF stage leads to the overall performance improvement. Next, we analyze the node scalability of the MGS stage and the CF stage respectively.

Node Scalability of Multi-grained Scanning: As mentioned in Section 2, the MGS stage is composed of two steps: window scanning and feature transformation. Note that the total amount of data produced by window scanning is usually much larger than that of the original input data. The scanning results with a specific scanning window will be used to train two random forests, and then the class vectors are generated and concatenated as transformed features. According to the experiment setting, three scanning windows of different sizes are used. Thus, six random forests will be trained in the MGS stage. Due to the window scanning, the size of the input data for each random forest becomes very large. Taking the sEMG dataset as an example. The original size of the sEMG dataset is 28.84 MB. The sizes of scanning results using three different windows are 4.94 GB, 9.24 GB and 15.84 GB respectively, each of which is hundreds of times of the original input.

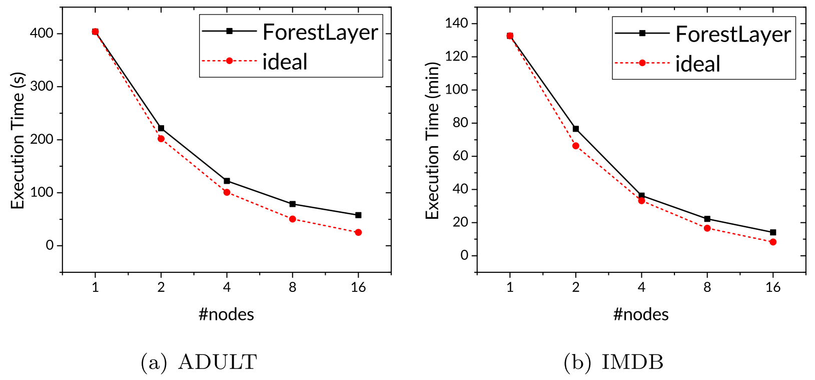
When only two nodes are used, the number of forests is larger than the number of nodes and thus each node needs to train three random forests. Moreover, the input of each random forest in each node may correspond to different scanning windows. Thus, if all forests in each node are trained in parallel, each node may need to store the scanning results of all scanning windows, which leads to huge memory overhead. Since the memory consumption may exceed the size of available memory, we limit the degree

Table 6
Performance of ForestLayer and tfForest in the cluster with 16 nodes.

Dataset	Time	tfForest			ForestLayer		
	Unit	Runtime	Accuracy	#cascade levels	Runtime	Accuracy	#cascade levels
ADULT	sec	1368	85.97%	12	**57.7**	**86.07%**	5
YEAST	sec	815.4	62.33%	1	**13.8**	**62.78%**	1
LETTER	sec	737.7	88.0%	2	**59.6**	**97.45%**	5
IMDB	min	104.88	85.65%	14	**14.1**	**89.13%**	15
sEMG	min	683.57	59.26%	1	**85.8**	**71.48%**	2
MNIST	min	**29.92**	96.16%	5	64.4	**99.33%**	18
CIFAR10	min	**125.32**	42.3%	21	164.9	**62.12%**	14

Table 7
Performance of ForestLayer and tfForest in the MGS and CF stages in minutes.

Dataset	tfForest			ForestLayer			MGS	CF	Overall
	MGS	CF	Overall	MGS	CF	Overall	Speedup	Speedup	Speedup
sEMG	**13.97**	669.6	683.57	84.47	**1.34**	**85.8**	0.17×	499.7×	**7.97×**
MNIST	**9.74**	20.18	**29.92**	44.8	**19.6**	64.4	0.22×	1.03×	0.46×
CIFAR10	**15.57**	109.75	**125.32**	98.29	**66.68**	164.9	0.12×	1.65×	0.76×



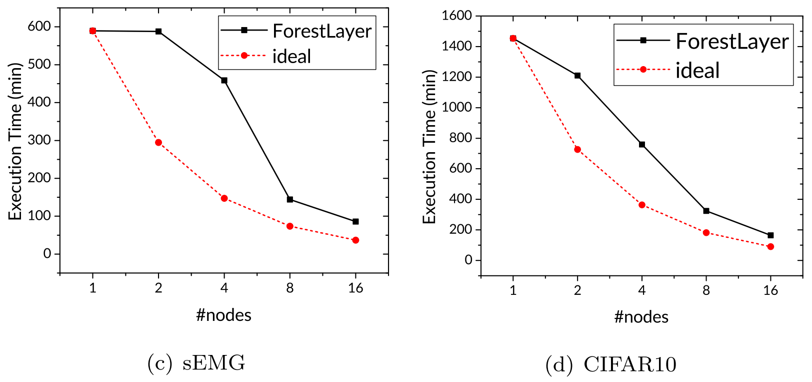


Fig. 7. Node scalability.

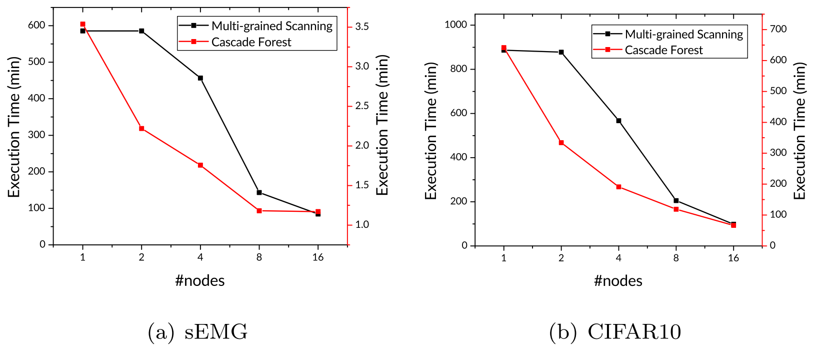


Fig. 8. Node scalability of multi-grained scanning and cascade forest.

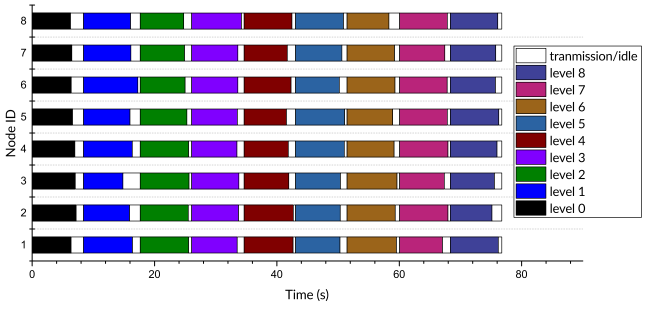


Fig. 9. Load balance. Dataset: ADULT. #nodes = 8. Split granularity = 125. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of computation parallelism during the MGS stage and all random forests are trained sequentially. Therefore, in the case of two nodes, the performance of the multi-grained scanning on the sEMG and CIFAR10 datasets is not improved significantly.

As the number of computing nodes increases, the number of forest training tasks allocated to each node becomes smaller. For example, in the case of four nodes, each node will train two random forests at most. Moreover, when the number of available nodes exceeds the total number of random forests, each random forest will be split into sub-forests using a given split granularity. For example, when scaling to 16 nodes, we can split each forest into three sub-forests. As a result, at most two sub-forest training tasks will be allocated to each node. Due to fewer forest training tasks in each node, the total memory overhead for storing the scanning results will be reduced and the memory overhead is no longer a performance bottleneck. Thus, we can utilize all computing nodes and train all forests in parallel. From Figs. 8(a) and 8(b), we can see that the scalability of the multi-grained scanning tends to be near-linear when scaling to more than two nodes.

Node Scalability of Cascade Forest: Compared with the forest in the MGS stage, the input size of the forest in each level of the cascade forest is much smaller. In addition, all forests in a cascade level have the same input data and thus all forest training tasks allocated to a node can share the same copy of input data. As a result, the memory consumption for storing the input data in each computing node is low and we can allocate more forest or sub-forest training tasks to each node. The forests in all nodes can be trained concurrently. Moreover, during the CF stage, smaller split granularity can be used, which means a forest will be split into more sub-forest and each sub-forest has fewer trees. Since more sub-forests are trained in parallel, the degree of computation parallelism can be improved significantly.

Due to the high degree of computation parallelism, the CF stage achieves better node scalability than the MGS stage. From Figs. 8(a) and 8(b), we can see that the CF stage achieves near-linear scalability no matter how many nodes are used. Note that for the sEMG dataset (Fig. 8(a)), the performance of the CF stage in the case of 16 nodes is not improved significantly. The main reason is that the total number of instance on CF stage for the sEMG dataset is small and the forest training overhead is relatively low. When scaling to 16 nodes, more sub-forests with less number of trees will be generated. Despite the high computation parallelism, the network communication overhead caused by the merging process of the sub-forest training results will also become large, which hinders the performance improvement in the case of 16 nodes.

Load balance. We also evaluate the load balance of ForestLayer on the first type of datasets in a cluster of 8 nodes. Fig. 9 illustrates the training time and the transmission/idle time of each cascade level in each node.

The colored squares denote the actual training time of each node in each cascade level. From Fig. 9, we can see that in each cascade level, the training time in each node is almost the same. Thus, ForestLayer achieves good load balance between all nodes.

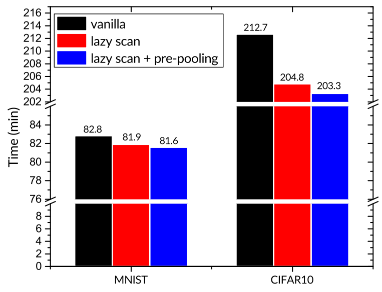
Specifically, the average training time for all nodes is 66.8 s. We compute the standard deviation, which is as low as 0.73% (0.49/66.8). In each cascade level, the standard deviation of the training time across the whole cluster is only 2.88%−8.16% of the average training time. These results validate the good load balance of ForestLayer.

5.5. Effect of system-level optimizations

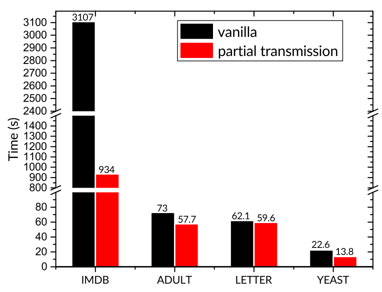
We evaluate the effectiveness of the proposed system-level optimization techniques. Fig. 10 demonstrates that the proposed optimization techniques are effective for the performance improvement.

The lazy scan optimization and the pre-pooling optimization are used during the MGS stage. To evaluate the two optimization techniques, we select the image datasets (i.e., MNIST and CIFAR10). As shown in Fig. 10(a), the two optimization techniques are effective for improving the efficiency of the multi-grained scanning. By employing both lazy scan and pre-pooling, ForestLayer runs up to 4.4% faster than the vanilla version.

Since the partial transmission optimization is only used in the cascade forest, we select datasets IMDB, ADULT, LETTER, and YEAST to evaluate the partial transmission optimization technique. As illustrated in Fig. 10(b), with the partial transmission technique, ForestLayer runs 4.02% − 70% faster than the version



(a) Lazy scan and pre-pooling. Y axis indicates the runtime of multi-grained s-canning. #node=8.



(b) Partial transmission. Y axis indicates the runtime of cascade forest. #node=16.

Fig. 10. Effectiveness of system-level optimization techniques.

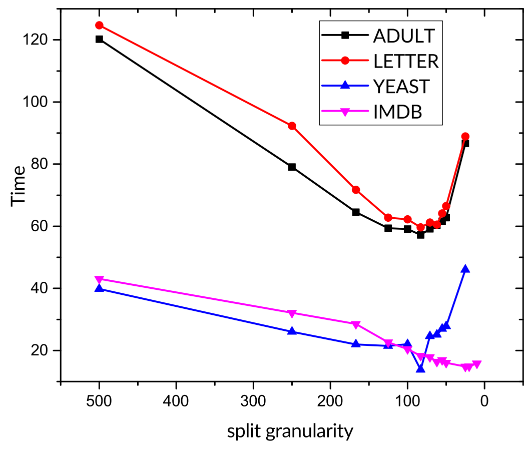


Fig. 11. The training time with varying split granularities. Time unit: sec (ADULT, LETTER, YEAST), min (IMDB). #node = 16.

without the partial transmission technique. The partial transmission optimization can reduce the network communication overhead significantly, especially for the high-dimensional datasets such as the IMDB dataset.

5.6. Analysis of the split granularity

The split granularity represents the number of trees in each sub-forest, which plays an important role in the overall performance. Thus, we also evaluate and analyze the effects of split granularity g on the overall performance. In addition, we empirically analyze the influence factors of the optimal split granularity.

Fig. 11 illustrates the training time of ForestLayer under different split granularities.

The training time decreases when decreasing the split granularity g. The split granularity determines the degree of computation parallelism. The smaller split granularity means that each sub-forest contains fewer trees. As the split granularity decreases, the total number of sub-forests will be larger. Thus, we can achieve higher degree of computation parallelism.

However, when the split granularity becomes very small, the training time begins to increase. For the low-dimensional datasets (i.e., ADULT, LETTER, and YEAST), when the split granularity is less than about 83, the training performance becomes worse. For the high-dimensional dataset IMDB, the critical value of the split granularity is about 20. The main reason is two-fold. First, by splitting forests into smaller sub-forests, more sub-forest training tasks will be generated. Thus, huge network communication overhead during the merging process of the sub-forest training results will be caused. Second, the task scheduling overhead will also increase due to more sub-forest training tasks.

In addition, compared to the low-dimensional datasets, the IMDB dataset has a smaller critical value of the split granularity (i.e., 20). Since the IMDB dataset is high-dimensional, the training time of each sub-forest is relatively large. Due to the higher computation-to-communication ratio, the IMDB dataset can hold much smaller split granularity.

Fig. 12 illustrates the training time distribution, which contains the total network transmission time, the total task synchronization time, and the total computation time.

First, as shown in Figs. 12(a) and 12(b), with the decrease of g, the total task synchronization time decreases. The fine-grained splitting leads to high computation parallelism. Thus, the load of each node is more balanced.

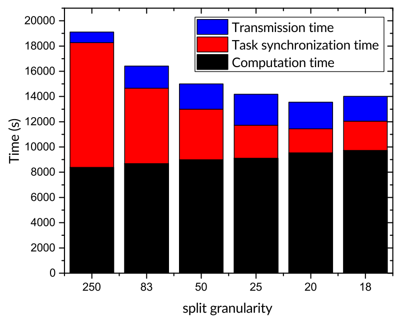
Second, as g decreases, the total transmission time grows. The smaller split granularity will lead to larger communication overhead, which causes a lower computation to communication ratio. Thus, when the split granularity exceeds a certain threshold, the performance becomes worse.

At last, as g decreases, the total computation time grows. Since the total number of trees is fixed, the computation load of training all sub-forests is fixed. In practice, the growing total computation time is caused by the scheduling overhead. The task scheduling overhead increases due to the large number of computing tasks. As illustrated in Fig. 12(a), the scheduling overhead accounts for a small proportion of the total computation time. The reason is that training process of the high-dimensional IMDB dataset involves larger computation load and thus requires more computation time. Thus, the scheduling overhead is not significant. But, as Fig. 12(b) shows, for the low-dimensional dataset (i.e., ADULT) that requires low computation overhead, the scheduling overhead accounts for a relatively large proportion of the computation time.

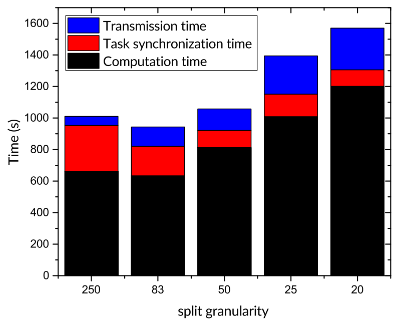
We further evaluate the relationship between the optimal split granularity and the number of cluster nodes. Fig. 13 illustrates the optimal split granularity when varying the number of nodes. As the number of nodes grows, the optimal granularity decreases. With more computing nodes, the degree of computation parallelism can be higher and each node can accommodate more fine-grained training tasks. Thus, we can set smaller task granularity when more computing nodes are available.

6. Related work

- *Multi-layered representation learning.* Representation learning is the key reason for the success of DNNs [5]. With a multi-layered deep structure, the model is able to learn a good representation from the raw input data. Recently, Zhou and Feng [40] proposed the deep forest, which is the first attempt to construct



(a) Dataset: IMDB.



(b) Dataset: ADULT.

Fig. 12. Time distribution under different split granularities. #nodes = 16.

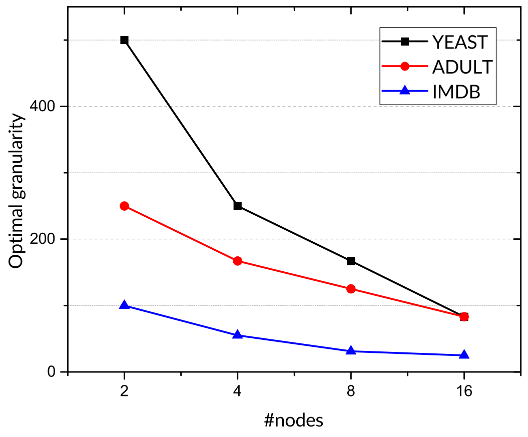


Fig. 13. Optimal split granularity with varying number of nodes.

a multi-layered forest ensemble to do representation learning. Deep forest performs highly competitive to DNNs. Moreover, deep forest has attracted more attention due to its representation learning ability, adaptive model complexity and robustness to hyper-parameter settings [33,36,38]. Zhang et al. [39] proposed a deep forest variant specifically for the detection of cash-out fraud.

- *Machine learning framework.* Many machine learning frameworks such as TensorFlow [1], Pytorch [26], MXNet [11], and Keras [14] are proposed and widely used in many applications. However, they are specifically designed for DNNs and thus not very suitable for deep forest. gcForest [18,40] is the first system for the training of deep forest. However, gcForest is neither very efficient nor scalable. Zhang et al. [39] implemented distributed deep forests based on the commercial computing framework KunPeng [41] for the purpose of detecting the cash-out fraud. They used KunPeng-MART as the base learner of deep forest. KunPeng employs a data-parallel approximate histogram algorithm to train MART. In this paper, we use the random forest as the base learner and propose a fine-grained sub-forest based task-parallel algorithm.

- *Declarative programming model.* Declarative programming is well studied and widely used in many machine learning frameworks. TensorFlow [1] and MXNet [11] use the computation graph to represent DNNs. Torch [15], Pytorch [26], and Keras [14] employ the layer abstraction to represent each module of DNNs. Inspired by these frameworks, we propose high-level declarative programming APIs. A deep forest is represented as a computation graph, including the multi-grained scanning layer and the auto-growing cascade layer.

7. Conclusion and future work

In this paper, we presented ForestLayer, an efficient and scalable deep forest system built on distributed task-parallel platforms. First, we proposed a fine-grained sub-forest based task-parallel algorithm. Then, to ensure the consistency of the training results with the non-splitting method, and at the same time minimize the expected training time, we proposed a novel uniform splitting mechanism. Next, three system-level optimization techniques, including lazy scan, pre-pooling, and partial transmission were proposed to further improve the performance of ForestLayer. We have implemented ForestLayer on the distributed task-parallel platform Ray with easy-to-use high-level programming APIs. Experimental results show that ForestLayer outperforms the existing deep forest system gcForest with $7\times$ to $20.9\times$ speedup on a range of datasets. In addition, ForestLayer outperforms TensorFlow-based implementation on most of the datasets, while achieving better predictive performance. Furthermore, ForestLayer achieves near-linear scalability and good load balance.

In the future, we plan to design an adaptive algorithm to find the optimal split granularity. We also plan to support more types of base learners. In addition, we plan to apply the deep forest to more real-world applications.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant Numbers 61572250, U1811461, 61702254), Jiangsu Province Science and Technology Support Program, China (Grant Number BE2017155), National Natural Science Foundation of Jiangsu Province, China (Grant Number BK20170651), the program B for Outstanding Ph.D. candidate of Nanjing University, Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu Province, China, and Project funded by China Postdoctoral Science Foundation.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.05.001.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI, vol. 16, 2016, pp. 265–283.
[2] Apache Hadoop, https://hadoop.apache.org.
[3] Apache arrow, http://arrow.apache.org/.
[4] K. Bache, M. Lichman, UCI Machine Learning Repository, 2013.
[5] Y. Bengio, A. Courville, P. Vincent, Representation learning: A review and new perspectives, IEEE Trans. Pattern Anal. Mach. Intell. 35 (8) (2012) 1798–1828.
[6] P. Berenbrink, T. Friedetzky, Z. Hu, R. Martin, On weighted balls-into-bins games, in: Conference on Theoretical Aspects of Computer Science, 2005, pp. 231–243.
[7] P. Berenbrink, T. Friedetzky, Z. Hu, R. Martin, On weighted balls-into-bins games, Theoret. Comput. Sci. 409 (3) (2008) 511–520, http://dx.doi.org/10.1016/j.tcs.2008.09.023.
[8] L. Breiman, Random forest, Mach. Learn. 45 (2001) 5–32.
[9] T. Chen, C. Guestrin, XGBoost: A scalable tree boosting system, in: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794, http://dx.doi.org/10.1145/2939672.2939785.
[10] J. Chen, K. Li, K. Bilal, X. Zhou, K. Li, P.S. Yu, A bi-layered parallel training architecture for large-scale convolutional neural networks, IEEE Trans. Parallel Distrib. Syst. 30 (5) (2019) 965–976, http://dx.doi.org/10.1109/TPDS.2018.2877359.
[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, arXiv preprint arXiv:1512.01274, 2015.
[12] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, K. Li, A parallel random forest algorithm for big data in a spark cloud computing environment, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2017) 919–933, http://dx.doi.org/10.1109/TPDS.2016.2603511.
[13] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, T. Li, Performance-aware model for sparse matrix-matrix multiplication on the sunway taihulight supercomputer, IEEE Trans. Parallel Distrib. Syst. 30 (4) (2019) 923–938, http://dx.doi.org/10.1109/TPDS.2018.2871189.
[14] F. Chollet, et al., Keras. https://keras.io.
[15] R. Collobert, S. Bengio, J. Mariéthoz, Torch: A Modular Machine Learning Software Library, Tech. rep., Idiap, 2002.
[16] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (2011) 2493–2537.
[17] T. Colthurst, D. Sculley, G. Hendry, Z. Nado, Tensorforest: scalable random forests on tensorflow, in: Machine Learning Systems Workshop At NIPS, 2016.

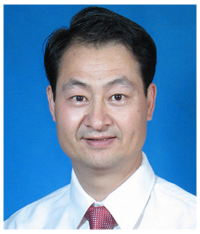
[18] gcForest version 1.0, 2014. http://lamda.nju.edu.cn/code_gcForest.ashx.
[19] P. Geurts, D. Ernst, L. Wehenkel, Extremely randomized trees, Mach. Learn. 63 (1) (2006) 3–42, http://dx.doi.org/10.1007/s10994-006-6226-1.
[20] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Computer Vision and Pattern Recognition, CVPR, 2016, pp. 770–778.
[21] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A. r. Mohamed, N. Jaitly, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, IEEE Signal Process. Mag. 29 (6) (2012) 82–97.
[22] A. Krizhevsky, Learning Multiple Layers of Features from Tiny Images, 2009.
[23] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, NIPS, 2012, pp. 1097–1105.
[24] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324, http://dx.doi.org/10.1109/5.726791.
[25] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, I. Stoica, Ray: A distributed framework for emerging AI applications, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2018, pp. 561–577.
[26] Pytorch, https://pytorch.org/.
[27] Ray homepage, 2018. https://rise.cs.berkeley.edu/projects/ray/.
[28] C. Sapsanis, G. Georgoulas, A. Tzes, D. Lymberopoulos, Improving EMG based classification of basic hand movements using EMD, in: 2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC, 2013, pp. 5754–5757, http://dx.doi.org/10.1109/EMBC.2013.6610858, issn=1094-687X.
[29] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556, 2014.
[30] I. Stoica, D. Song, R.A. Popa, D. Patterson, M.W. Mahoney, R. Katz, A.D. Joseph, M. Jordan, J.M. Hellerstein, J.E. Gonzalez, et al., A Berkeley view of systems challenges for AI, 2017, arXiv preprint arXiv:1712.05855.
[31] SysML Conference, 2014. http://www.sysml.cc/.
[32] Tensorforest example, https://github.com/aymericdamien/TensorFlow-Examples/.
[33] L.V. Utkin, M.A. Ryabinin, A Siamese deep forest, arXiv preprint arXiv:1704.08715, 2017.
[34] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., Google's neural machine translation system: Bridging the gap between human and machine translation, arXiv preprint arXiv:1609.08144, 2016.
[35] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, G. Zweig, The microsoft 2016 conversational speech recognition system, in: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, 2017, pp. 5255–5259.
[36] C.-Y. Yu, Z.-M. Qi, H. Liu, Sound event detection using deep random forest, in: Detection and Classification of Acoustic Scenes and Events, 2017.
[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Usenix Conference on Networked Systems Design and Implementation, NSDI, 2012, p. 2.
[38] C. Zhang, J. Yan, C. Li, R. Bie, Contour detection via stacking random forest learning, Neurocomputing 275 (2018) 2702–2715.
[39] Y.-L. Zhang, J. Zhou, W. Zheng, J. Feng, L. Li, Z. Liu, M. Li, Z. Zhang, C. Chen, X. Li, et al., Distributed deep forest and its application to automatic detection of cash-out fraud, arXiv preprint arXiv:1805.04234, 2018.
[40] J.F. Zhi-Hua Zhou, Deep forest: Towards an alternative to deep neural networks, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI, 2017, pp. 3553–3559, http://dx.doi.org/10.24963/ijcai.2017/497.
[41] J. Zhou, X. Li, P. Zhao, C. Chen, L. Li, X. Yang, Q. Cui, J. Yu, X. Chen, Y. Ding, Y.A. Qi, KunPeng: Parameter server based distributed learning systems and its applications in Alibaba and Ant financial, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 1693–1702, http://dx.doi.org/10.1145/3097983.3098029.

Guanghui Zhu received the MS degree in Southeast University, China, in 2012. He is currently working towards the Ph.D. degree in Nanjing University. His research interests include parallel computing, distributed systems.

Qiu Hu received the BS degree in University of Electronic Science and Technology of China in 2016. He is currently working towards the Master degree in Nanjing University. His research interests include parallel computing, distributed systems.

Rong Gu is an assistant researcher at State Key Laboratory for Novel Software Technology, Nanjing University, China. Dr. Gu received the Ph.D. degree in computer science from Nanjing University in December 2016. His research interests include parallel computing, distributed systems and distributed machine learning.

Chunfeng Yuan is currently a professor in computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. She received her bachelor and master degrees in computer science from Nanjing University. Her main research interests include computer architecture, parallel and distributed computing and information retrieval.



Yihua Huang is currently a professor in computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. He received his bachelor, master and Ph.D. degrees in computer science from Nanjing University. His main research interests include parallel and distributed computing, big data parallel processing, distributed machine learning algorithm and system, and Web information mining.