Cui *et al. Journal of Cloud Computing* (2023) 12:22
https://doi.org/10.1186/s13677-022-00382-7

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

A bidirectional DNN partition mechanism for efficient pipeline parallel training in cloud

Lingyun Cui[1], Zhihao Qu[1*], Guomin Zhang[2*], Bin Tang[1] and Baoliu Ye[3]

Abstract

Recently, deep neural networks (DNNs) have shown great promise in many fields while their parameter sizes are rapidly expanding. To break through the computation and memory limitation of a single machine, pipeline model parallelism is proposed for large-scale DNN training by fully utilizing the computation and storage power of the distributed cluster. Cloud data centers can also provide sufficient computing, storage and bandwidth resources. However, most existing approaches apply layer-wise partitioning, which is difficult to obtain an even model partition result because of the large computational overhead discrepancy between DNN layers, resulting in degraded efficiency. To tackle this issue, we propose "Bi-Partition", a novel partitioning method based on bidirectional partitioning for forward propagation (FP) and backward propagation (BP), which improves the efficiency of the pipeline model parallelism system. By deliberated designing distinct cut positions for FP and BP of DNN training, workers in the pipeline get nearly equal computational loads, and the balanced pipeline fully utilizes the computing resources. Experiments on various DNN models and datasets validate the efficiency of our mechanism, e.g., the training efficiency achieving up to 1.9× faster than the state-of-the-art method PipeDream.

Keywords Pipeline model parallelism, Deep neural network, Distributed system, Cloud data center

Introduction

The last decade has witnessed an unprecedented booming of deep neural networks (DNNs) in many applications such as image recognition [1, 2], intelligent speech [3], and machine translation [4]. Nowadays, widely-used DNN models usually consist of tens to hundreds of layers with millions to billions of parameters. For example, in natural language processing, the number of parameters in modern DNN models has increased from 8.3 billion parameters (Megatron-LM [5], 2019) to 1600 billion parameters for Switch transformers [6] in 2021.

DNNs with large model size greatly improve the inference accuracy and generalization capability, but the training tasks are computation-intensive and require numerous data samples. With virtualization [7, 8] and resource scheduling technologies [9, 10], cloud data centers can manage rich computing, storage, and bandwidth resources in clusters. To break through the limitation of hardware capacity on a single machine in the cluster, data parallelism (DP) and model parallelism (MP) are two mainstream approaches to training large-scale DNNs over distributed workers [11–17]. In DP, training data are partitioned and distributed to workers for local training. Since each worker must maintain a replica of the DNN model, memory constraint is still unsolvable for large-scale DNN training. In MP, the model partition algorithm splits the DNN model and deploys them to each device in the cloud data center as shown in Fig. 1. In this way, the computing and storage load of the large-scale DNN model can be

*Correspondence:
Zhihao Qu
quzhihao@hhu.edu.cn
Guomin Zhang
40519667@qq.com
[1] Key Laboratory of Water Resources Big Data Technology of Ministry of Water Resources and School of Computer and Information, Hohai University, Nanjing, China
[2] Army Engineering University, Nanjing, China
[3] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

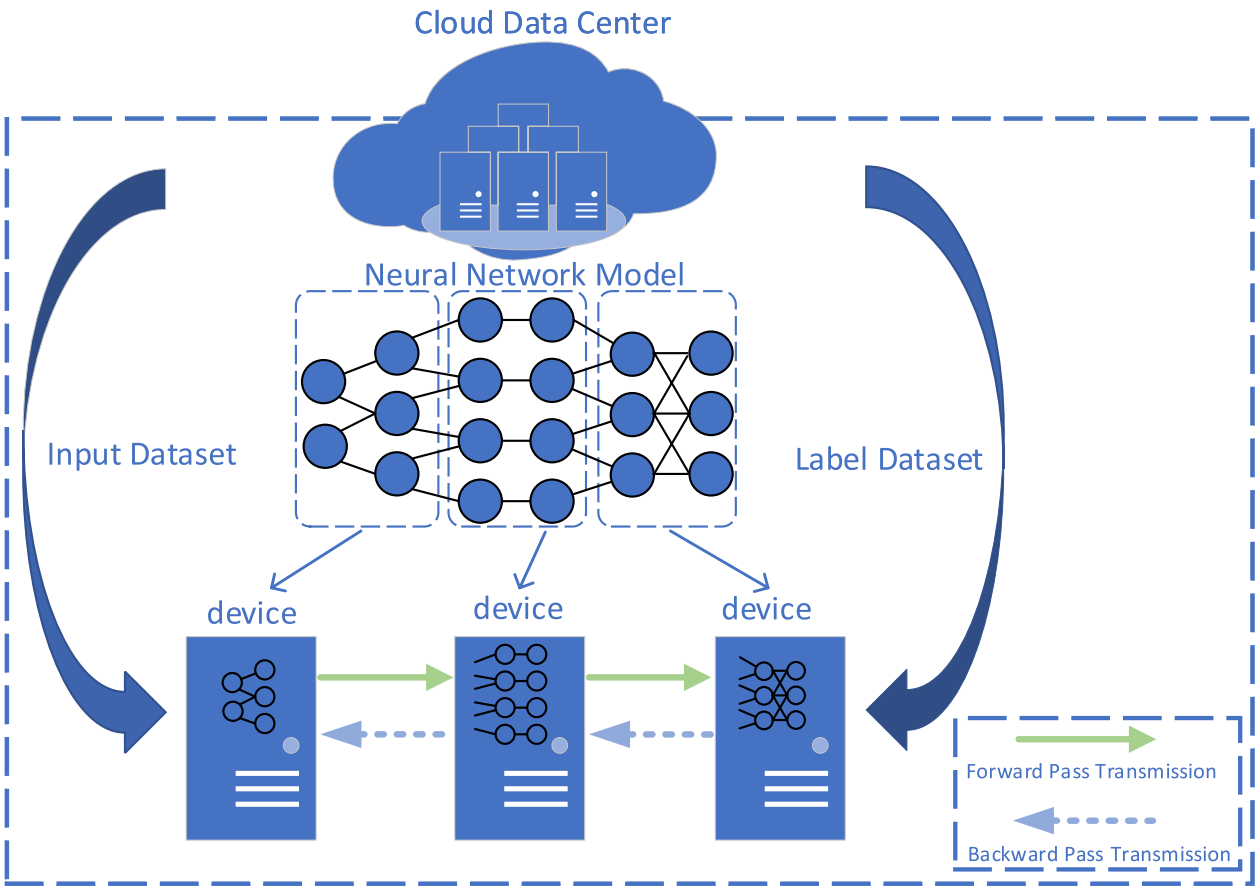


Fig. 1 Model parallelism (MP) in cloud

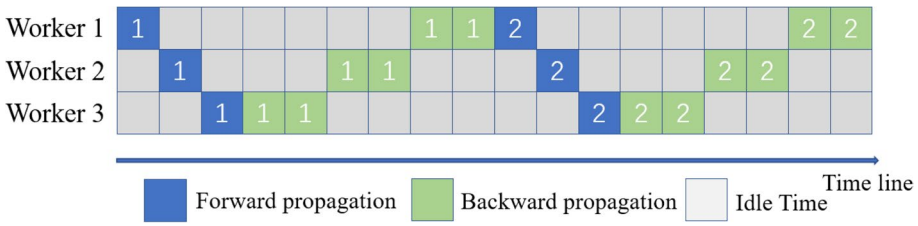
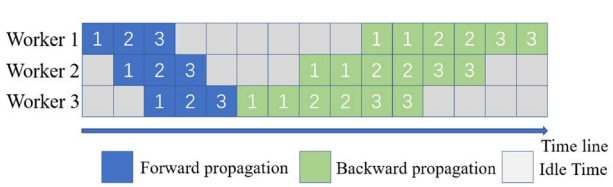


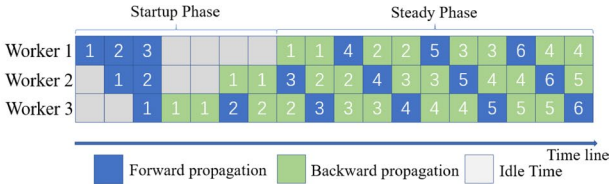
Fig. 2 An example of three workers training under traditional MP. For simplicity, the time cost of BP is twice that of FP. Basically, this assumption is consistent with the characteristic of wall-clock time cost for model training in realistic scenarios. The BP process of a DNN layer consists of two steps: calculating the error/gradient of the current layer based on the error/gradient passed from the successive layer, and updating the model parameters of the current layer. On the contrast, the FP process of a DNN layer only calculates the output value for the next layer, which is generally have the same complexity as calculating the error/gradient. Therefore, the time consumption of BP is generally considered to be about twice that of FP

distributed to the cluster of the cloud data center. As illustrated in Fig. 2, three workers with partitions of a DNN model collaborating to train a complete DNN model. Numbers in cubes denote the training sample/batch ID, and blue and green cubes indicate forward propagation (FP) and backward propagation (BP), respectively. In traditional MP approaches, despite feasible costs in terms of memory for each worker, computation resources are underutilized because training on the successive worker should wait for the training results of the current worker. As shown in Fig. 2, the idle cubes, namely "bubbles", severely degrade the efficiency of the DNN training.

To address this problem, pipeline model parallelism (PMP) techniques [18–23] have been proposed to improve the resource utilization of MP by pipeline-like scheduling. Most of these works are devoted to



(a) Micro-batch pipeline model parallelism.



(b) One forward one backward (1F1B) pipeline model parallelism.

Fig. 3 Two mainstream techniques of pipeline model parallelism

improving the efficiency of computational parallelism in the PMP system through different pipeline parallelism strategy tuning.

Although significant advances have been made in PMP techniques, all of them suffer from the problem of uneven DNN model partitioning i.e., workers take different time to process the partitions of DNN allocated to them. It is recognized that the training efficiency of a pipeline is limited by the slowest worker. The traditional layer-wise partitioning scheme cut the DNN into several partitions consisting of sequential layers. However, it is hard to partition the DNN to achieve perfect balance in terms of computation time because BP and FP on different layers have significantly diverse computation overheads. To address this problem, this paper proposes "bi-partition", a bidirectional model partitioning method that partitions the FP and BP of a DNN separately. Bi-partition allows diverse partitions on the FP and BP of a DNN layer to achieve an even partition scheme, such that BP and FP of a DNN layer may be separated at different workers. The layer-wise partitioning approach is a particular case of the bi-partition approach when forward and backward computations of all layers are not separated. Notably, fine-grained bi-partition can allocate the training time on each worker evenly and improve the training efficiency of the pipeline system.

The main contributions of this paper can be summarized as follows.

- We propose bi-partition, a novel bidirectional model partitioning mechanism that alleviates the uneven partitioning of the traditional layer-wise model partitioning strategy and improves the training efficiency of pipeline model parallelism.
- Aiming to minimize the execution time of the pipeline, we formulate the problem of partitioning DNN model and propose an efficient algorithm based on dynamic programming, which can obtain the optimal model partition scheme in polynomial time. Moreover, we find the monotonicity of its sub-problem and use the bisection method to further reduce the complexity of the algorithm.
- Extensive experiments are conducted on various DNN models and datasets. The results demonstrate that the training efficiency of our approach is 1.9 × faster than the state-of-the-art PipeDream [20].

This paper is organized as follows: Related work section summarizes and discusses existing methods of PMP. System model section describes the architecture of our proposed system. Bi-partition method section shows how the bi-partition strategy works on pipeline model parallelism. Implementation section and Performance evaluation section show the implementation details of our systems and validates the proposed methods on various DNN models and datasets. Finally, Conclusion section concludes this paper with an outlook. The code has been open-sourced to Github[1]

Related work

DNNs are composed of layers of computational units. Each layer of the DNN records its model parameters in a multidimensional tensor, and the size of these model parameters varies with each layer of computation. When training a large-scale DNN, PMP partitions the DNN model and allocates different parts to different available workers, thus avoiding that a single machine cannot afford the memory overhead. Applying pipelined scheduling to train DNNs in parallel can greatly improve computational efficiency of the system. In this following, we will introduce some representative PMP methods and discuss the key challenges in PMP.

There are two mainstream techniques of PMP to reduce "bubbles" in the pipeline as Fig. 3a and b showing: (1) micro-batch, (2) one forward one backward (1F1B). Gpipe [18] is the first to introduce the micro-batch, which splits the mini-batch into smaller equal

[1] https://github.com/cccccly/Parallel-SGD

micro-batches and injects them into the pipeline concurrently to reduce the "bubbles" of the distributed system. To further reduce the "bubbles", Chimera [19] introduces bidirectional pipelines, which combine two pipelines in different directions. Both the pipelines hold a complete DNN model and apply the micro-batch technique. Chimera can fill the idle time of a single pipeline and reduce the number of "bubbles" by up to 50% than Gpipe. Due to the limitations of the synchronous updating method, the efficiency of the pipeline systems based on micro-batch technology still deteriorates because of inevitable "bubbles". PipeDream [20] proposes 1F1B, which supports asynchronous updates to alleviate the "bubbles" problem in micro-batch-based approaches. In the pipeline, workers under 1F1B scheduling perform a backward propagation immediately after a forward propagation (the two computations use different data samples/mini-batches). When the pipeline is absolutely balanced, 1F1B ensures the pipeline is free of "bubbles".

In addition to the study of pipeline parallel strategy method, storage, convergence performance and practicality of pipeline model parallel system are also explored. PipeDream-2BW [21] mainly solves the storage problem of PipeDream by using double buffered gradients to alleviate the storage overhead caused by a large number of gradient copies in PipeDream. It also combines the activation recomputation technique to further reduce the memory consumption of activation values. vPipe [24] dynamically adjusts the model partitioning and memory management during the training of neural networks, and fully adapts to the dynamics in training by an online algorithm to find the near-optimal model partitioning when training dynamic neural networks and neural architecture search (NAS). DAPPLE [22] is a synchronous training framework that combines data parallelism and pipeline parallelism to improve computational efficiency while ensuring convergence. The goal of HetPipe [23] is to allow heterogeneous GPUs to form an architecture to collaboratively train large-scale neural networks that cannot be trained by a single GPU. It forms several groups of heterogeneous GPUs, with pipelined model parallelism within the group to make full use of efficient link resources such as the Nvlink, and data parallelism between the groups to make full use of computational resources.

Although PMP improves the training efficiency of large-scale DNNs, the unbalanced computational load over workers still hinders the computational efficiency when the model segments cannot be deployed uniformly to each node. The layer-wise partition approach is applied to most of the pipelined parallelism systems, but the computational load of each layer in a DNN varies greatly, making it difficult to produce a balanced loaded model partition result. Our bi-partition approach separately partitions the FP and BP in DNN training, allowing forward and backward propagation of neural network layers to be executed at different nodes. The separation of FP and BP in the computation intensive layer can alleviate the problem of excessive computational load on a single device. Eventually, the bi-partition method can improve the resource utilization of each node and improve the training efficiency of the PMP system.

System model

To fully utilize computing and memory resources, we implement a PMP system that uses a new partition method. Through the algorithm, we get a load-balanced model partition result, which is composed of forward and backward computation partition results. In the pipeline execution module, the load-balanced model partition result reduces the execution time of the slowest worker under the constraint of bandwidth and memory.

The architecture of our system is composed of three modules as Fig. 4 showing: (1) Profile Module: measuring the DNN model to get relevant data: computation time, activation size and weight size. (2) Model Bi-partition Module: using the relevant data and resource constraints to derive the optimal model partitions by the partition algorithm and allocate these partitions to each worker. (3) Pipeline Model Parallelism Module: scheduling workers to train a DNN model.

Profile module

Given a target DNN model, we use a profile program to train it on a single worker of the cluster for a short time (a few minutes). The profile program measures the running DNN training to record four data for each layer l. (1) T_l^f, the forward computation time of layer l (2) T_l^b, the backward propagation time of layer l, (3) w_l, the parameter size of layer l (4) a_l, the activation size of layer l. The preceding data are averaged for the batch size before ending the profile program.

Model bi-partition module

This module gets the optimal model partition scheme by the bi-partition algorithm. The goal is to minimize the pipeline model parallel execution time. At the same time, to ensure that the final obtained model partitioning result is valid, we must remove the invalid partitioning results under the resource constraints in terms of device memory. Finally, workers are assigned the corresponding partial models representing two results of forward computation partition result and backward computation partition result.

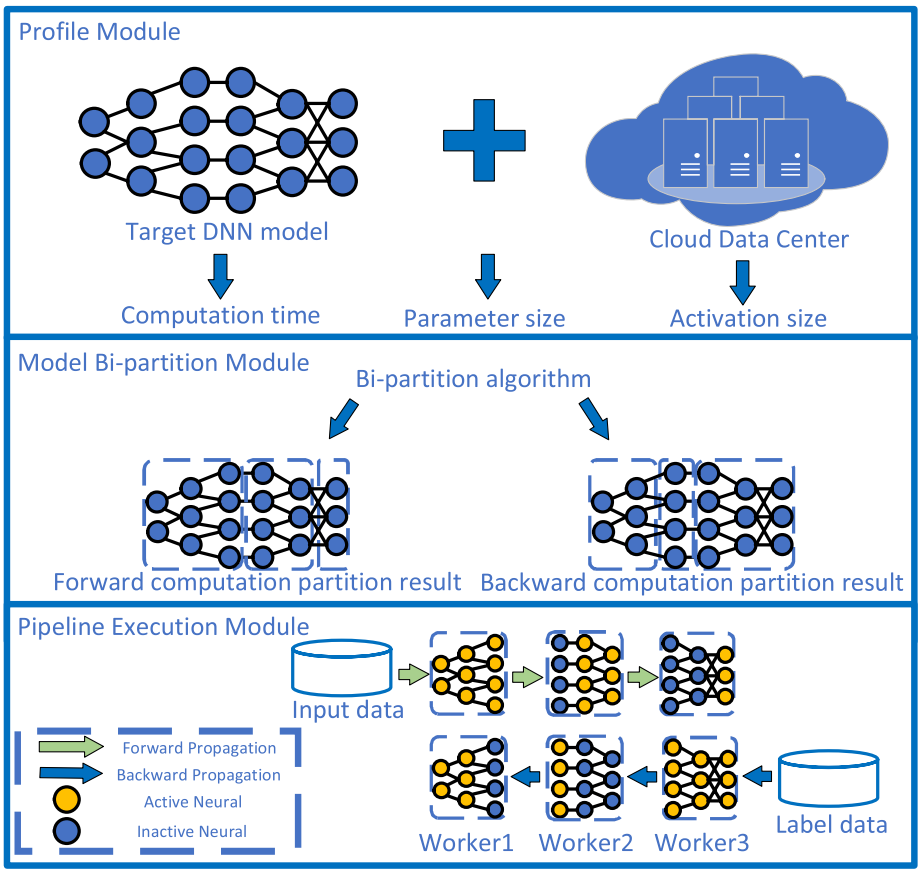


Fig. 4 System architecture

Pipeline execution module

The pipeline execution module is mainly responsible for the computation and communication schedule in the collaborative training model of each worker. According to the optimal partition result of the bi-partition algorithm, each worker is assigned a DNN model partition. The input layer uses the input data for forward propagation and the last layer uses the label data to calculate the loss, and starts backward propagation. Workers use the 1F1B strategy cycle to perform forward and backward propagation. Because the forward and backward computations are partitioned separately, there may be layers that only participate in partial computations on a worker, and these layers can only be performed once in the forward or backward direction.

Bi-partition method

Bi-partition overview

Bi-partition separately partitions the FP and BP, and each worker gains a range of them respectively by *bi-partition algorithm*. We consider a DNN model consisting of L layers, and for each layer l, FP is represented by F_l and BP by B_l. Let D be the set of workers, and each D_i has FP partition from F_{i_1} to F_{j_1}, and BP partition from B_{i_2} to B_{j_2}. The FP and BP partition for D_i means their training task in the pipeline execution module. Besides, there is no overlapping for the FP and BP partitions ranges and the two ranges for each worker both after the previous worker and before the latter worker. i.e.,

$$\begin{aligned} F_{(j-1)_1} < F_{i_1} <= \quad F_{j_1} < F_{(i+1)_1} \\ B_{(j-1)_2} < B_{i_2} <= \quad B_{j_2} < B_{(i+1)_2} \end{aligned} \tag{1}$$

$F_{(j-1)_1}$ and $B_{(j-1)_2}$ represented the end of the range of FP and BP for previous worker D_{i-1}, respectively. Notations $F_{(i+1)_1}$ and $B_{(i+1)_2}$ represented the start of the range of FP and BP for the successive worker D_{i+1}.

When $i_1 = i_2$ and $j_1 = j_2$ on all workers, the bi-partition approach is the same as the traditional layer-wise approach with identical partition. Therefore, we conclude that the traditional method is a special case of the bi-partition method. The bi-partition method can get a more balanced pipeline than traditional methods and thus improves the training efficiency of the pipeline parallelism system.

Here, we present the motivation for this work. Figures 5 and 6 illustrate that our bi-partition method outperforms traditional partition with the same scheduling 1F1B. We

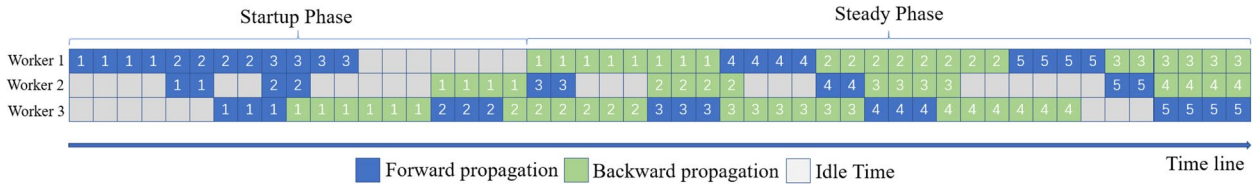


Fig. 5 An example of traditional layer-wise partition with 1F1B

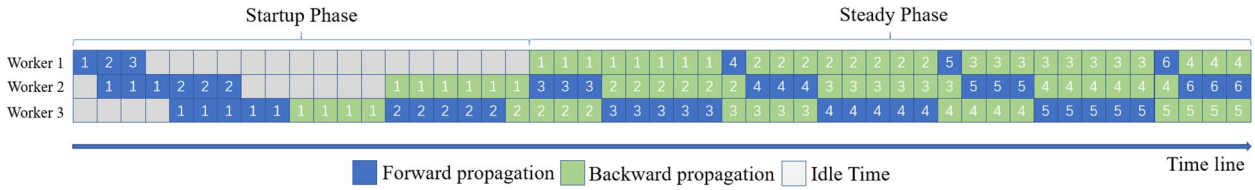


Fig. 6 An example of bi-partition with 1F1B

consider a DNN model that consists of four neural network layers which will be assigned to three workers. The forward computation time of one batch of data (one sample or one mini-batch) for each layer is F = {1, 3, 2, 3} and the backward computation time of one batch of data is B = {2, 6, 4, 6}. Let D_1, D_2, and D_3 denote the workers 1, 2, and 3, respectively. Under the traditional layer-wise partition algorithm, D_1 has layers 1-2, D_2 has layer 3, and D_3 has layer 4. When executing the training task, each worker runs all forward and backward computations for the owned layer. Since the 1F1B strategy keeps each batch of data training as a cycle during the steady phase, we can use the computation time of one batch of data to measure of the execution time of the whole pipeline. Thus, the execution time of each worker is {12, 6, 9}, respectively. Finally, since the execution time of the pipeline is limited by the slowest worker, the training time of the pipeline is 12. In Fig. 5, after the startup phase, the pipeline is bounded by the slowest worker D_1, causing resource underutilization on both D_2 and D_3. Figure 6 illustrates the training process under bi-partition mechanism, and all workers are free of idle time during the steady phase. The reason is that the DNN can be partitioned evenly by using bi-partition, where D_1 has $\{F_1\}$ and $\{B_1, B_2\}$; D_2 has $\{F_2, F_3\}$ and $\{B_3\}$; and D_3 has $\{F_4\}$ and $\{B_4\}$ as shown in Fig. 7. The BP of layer 2 is assigned to D_1, while its FP is assigned to D_2. The execution time on each worker can be treated as 9, because each worker has equal computational loads of 9. At this time, the pipeline reaches a state of computational balance and significantly improves the computational efficiency.

The above example shows the inherent shortcoming of the traditional layer-wise partition strategy. For computationally intensive DNN layers, both the FP and BP bring a huge computing overhead. The identical layer-wise partition scheme usually can not orchestrate layers to achieve a consistent load among workers because BP cost and FP cost are related in each worker, limiting the training efficiency. Our bi-partition method allows the FP and BP of these computationally intensive layers to be split, thus achieving an even partition result and fully utilizing the computational resources in the pipeline.

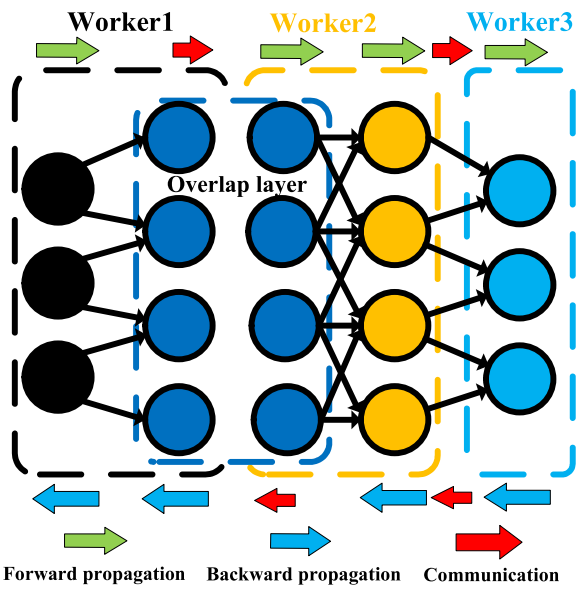


Fig. 7 An example for three workers with bi-partition

Bi-partition algorithm

Dynamic programming equations

To solve the bi-partition problem, the core idea is to find the dynamic programming equation that establishes the

relationship between the overall optimality and the optimality of the sub-problems. Let $A(i_1, j_1, i_2, j_2, m)$ denote execution time among m workers in the optimal pipeline across F_{i_1} to F_{j_1} for FP and B_{i_2} to B_{j_2} for BP. Term X formulates in (3).

$$A(i_1, j_1, i_2, j_2, m) = \min_{i_1 \le s_1 \le j_1} \min_{i_2 \le s_2 \le j_2} \min_{1 \le m' < m} X \tag{2}$$

The optimal pipeline can be cut into two optimal sub-pipelines by $\{s_1, s_2, m'\}$. Thus, we can solve the optimization problem using the optimal sub-problem property. We have:

$$X = \max \begin{cases} A(i_1, s_1, i_2, s_2, m - m'), \\ (a_{s_1} + a_{s_2})/B, \\ A(s_1 + 1, j_1, s_2 + 1, j_2, m'), \end{cases} \tag{3}$$

where the first term inside the max is the execution time of the first optimal sub-pipeline range from F_{i_1} to F_{s_1} for FP and B_{i_2} to B_{s_2} for BP with $m - m'$ workers, the second term is the time taken to communicate the activation of a_{s_1} and the gradients of a_{s_2} over a link with bandwidth B, and the third term is the execution time of the second optimal sub-pipeline range from F_{s_1+1} to F_{j_1} for FP and B_{s_2+1} to B_{j_2} for BP with m' workers.

The minimum execution time of the $A(i_1, j_1, i_2, j_2, m)$ also depends on to the result of the partition performed by the forward ($i_1 \to j_1$) and backward ($i_2 \to j_2$) propagation of the range using m workers. Therefore, the final goal of the algorithm is to find the minimum execution time of the $A(1, L, 1, L, M)$, which also corresponds to the result of the optimal model partitioning scheme.

Algorithm 1: Bi-partition Algorithm

Input: For each layer l: T_l^f, T_l^b, w_l, a_l. The bandwidth B. The memory capacity: C. The worker number: M
Output: Minimal pipeline execution time: A. Optimal partition: P

```
/* initial A when m = 1                                   */
for each legal U(i1, j1, i2, j2, 1) do
    A(U) = T(U)
for U_i in each legal U(i1, j1, i2, j2, m) do
    A(U_i) = ∞ // initial A(U_i)
    for S_i in each legal S(s1, m') do
        s2 = BSA(i1, i2, j1, j2, s1, m, m')
        /* update optimal pipeline A(U_i)                 */
        if Mem(U1) or Mem(U2) out of memory then
            continue // memory exceeded, skip this result
        if A(U_i) < max(A(U1), A(U2), (a_s1 + a_s2)/B) then
            A(U_i) = max(A(U1), A(U2), (a_s1 + a_s2)/B))
            P(U_i) = (s1, s2, m') // partition result
return A, P
```

Optimality

The theoretical analysis of its optimality can be divided into three steps. First, we give an abstract description of the optimal problem. In this paper, we denote the minimum pipeline execution time on m machines as a five-tuple $A(i_1, j_1, i_2, j_2, m)$, which indicates the minimal time of performing forward propagation over layers ($i_1 \to j_1$) and backward propagation over layers ($i_2 \to j_2$) using m nodes. Second, $A(i_1, j_1, i_2, j_2, m)$ can be partitioned into two sub-problems $A1(i_1, s_1, i_2, s_2, m - m')$ and $A2(s_1 + 1, j_1, s_2 + 1, j_2, m')$ by introducing a triple $S(s_1, s_2, m')$, where s_1, s_2 are cut layers for forward and backward propagation, respectively. Thus, $A(i_1, j_1, i_2, j_2, m)$ can be derived as the maximum of $A1(i_1, s_1, i_2, s_2, m - m')$, $A2(s_1 + 1, j_1, s_2 + 1, j_2, m')$, and the communication time between the two pipelines represented by $A1(i_1, s_1, i_2, s_2, m - m')$ and $A2(s_1 + 1, j_1, s_2 + 1, j_2, m')$, as shown in Eq. (3). Third, if we guarantee the optimality of the two sub-problems $A1(i_1, s_1, i_2, s_2, m - m')$ and $A2(s_1 + 1, j_1, s_2 + 1, j_2, m')$, then $A(i_1, j_1, i_2, j_2, m)$ is optimal because all triples $S(s_1, s_2, m')$ are checked by exhaustive searching, as shown in Eq. (2). Besides, the base case, i.e., any $A(i_1, j_1, i_2, j_2, 1)$ is unique and optimal when m is 1. Then, the optimality is also guaranteed by deriving from the base cases to the final objective $A(1, L, 1, L, M)$. Notably, at the time of each $A(i_1, j_1, i_2, j_2, m)$, we record the triple $S(s_1, s_2, m')$ that minimizes $A(i_1, j_1, i_2, j_2, m)$ as the optimal model partitioning scheme, which can be used to construct the optimal partition on m nodes.

Details of the Bi-partition algorithm

The pseudocode of the algorithm is shown as Algorithm 1. Let a 5-tuple $U(i_1, j_1, i_2, j_2, m)$ represent a training task across FP from F_{i_1} to F_{j_1} and BP from B_{i_2} to B_{j_2} among m workers ($i_1 > j_1$ and $i_2 > j_2$ cannot be satisfied at the same time). Let $T(U)$ denote the execution time taken by a single worker and $A(U)$ denote the execution time taken by the slowest worker among m workers in the optimal pipeline across range of U. $P(U)$ records the partition result corresponding to $A(U)$, which is represented by a triple $S(s_1, s_2, m')$. $Mem(U)$ denotes the memory cost across the range of U. We solve this optimization problem $A(U)$ based the results of two optimal sub-problems, $A(U_1)$ and $A(U_2)$, from $m = 1$ to $m = M$.

BSA

Algorithm 2 is to find the optimal s_2. We utilize the monotonicity of $A(U_1)$ and $A(U_2)$ with respect to s_2. $A(U_1)$ is monotonically non-decreasing, and $A(U_2)$ is non-increasing as s_2 increases, because the former computation increases and the latter decreases, but the number of computing devices remains the same. Thus, we can use the binary search for optimal s_2 over the initial searching region $[l_{s_2}, r_{s_2}]$. When the binary search iteratively contracts the upper and lower bounds until it stops, the optimal s_2 can be obtained between l_{s_2} and

$l_{s_2}+1$ by one comparison. This binary search reduces the complexity from $O(L)$ to $O(\log_2 L)$.

Algorithm 2: Binary Search Algorithm (BSA)

Input: $i_1, i_2, j_1, j_2, s_1, m, m'$
Output: optimal s_2
/* binary search to get the optimal s_2 */
$l_{s_2} = i_2 - 1, r_{s_2} = j_2$ // search region
while $l_{s_2} < r_{s_2}$ **do**
  $s_2 = (l_{s_2} + r_{s_2} + 1)/2$
  // two optimal sub-pipeline U_1 and U_2
  $U_1 = U(i_1, s_1, i_2, s_2, m - m')$
  $U_2 = U(s_1 + 1, j_1, s_2 + 1, j_2, m')$
  if $A(U_1) <= A(U_2)$ **then**
    $l_{s_2} = s_2$
  else
    $r_{s_2} = s_2$
find optimal s_2 between l_{s_2} and $l_{s_2} + 1$
return s_2

Runtime analysis

The complexity of the number of sub-problems is composed of five dimensions of $A(i_1, j_1, i_2, j_2, m)$. The algorithm needs to start from the base case of the number of devices 1 ($m = 1$). The range of forward and backward propagation represented by each legal ($i1$, $i2$, $j1$, $j2$) needs to be traversed once from the interval [1, L]. It stops when the number of devices is reached M. The complexity of solving each sub-problem is also that of traversing a pipeline consisting of all possible sub-pipelines, all legitimate $S(s1, s2, m')$. Therefore, the time complexity of the sub-problem solution should be $O(L^2 * m)$ before adding the BSA, and after adding it, the traversal of S_2 is changed to a binary search, so the complexity ends up being $O(L * \log_2 L * m)$. Finally, the complexity of the whole algorithm should be represented by the number of sub-problems and the product of the complexity of individual sub-problems $O(L^5 * \log_2 L * m^2)$.

Currently, the frequently-used DNNs consist of at most hundreds of layers, i.e., about 10-100 layers in VGG and 18-152 layers in ResNet. Since the partition algorithm only executes once, the complexity is acceptable. In our experiments, the running time is under 28 seconds.

Pipeline parallelism

After obtaining the partition results for each worker by the Bi-partition algorithm, each worker follows the sequence of data flow for training the DNN model. The FP output of the former worker will be used as input for the latter one, and the BP result of the latter worker will be used as input for the former worker. In the startup phase, we inject the number of workers of the batches as input data into the pipeline. In the steady phase, we hold the number of batches, and each worker achieves a full load when the pipeline is absolutely balanced.

Figure 3b shows a pipeline parallel with three workers in 1F1B scheme. In the startup phase, we inject three batches into the pipeline one by one, and D_i run BP after finishing $batche_{m-i+1}$ for FP (m denotes the number of workers). For all workers, once it has completed its FP of $batch_x$, the output is sent asynchronously to the next worker while simultaneously starting the BP of $batch_{x-m+i}$. Due to the overlaps between computation and communication, it can be seen that when the pipeline injects three batches simultaneously, each worker of the pipeline has no "bubbles" after the preparation time.

In bi-partition, each worker is assigned a forward propagation range and a backward propagation range. The FB and BP of the same layer may occur on different workers. It is recognized that the activation of FP needs to be provided to BP to calculate the gradient and update the model. Besides, the FP also depends on the updated model of the BP. Therefore, we need to send the activation values and updated model parameters to each other when the FP and BP run on the different workers.

Remarks

(1) *Communication and Computation Overlap:* When we communicate the activation and model parameters of calculation separate layers, the next layer's computational task runs asynchronously. Thus, the overhead of communication can be ignored when computation overlaps communication. (2) *Partial Copy*: To ensure the training convergence of 1F1B, each worker has multiple weight copies to overcome the staleness gradient. When the FP and BP of a layer are separated into two workers, the FP owner only stores a copy of the latest weight and sends its activation to the corresponding BP owner. Therefore, these separated layers only cost little memory overhead to the FP owner. (3) *Balanced Memory*: In PipeDream, the former workers require more weight copies, leading the first worker to suffer the most serious memory burden. The storage capacity of the first worker bounds other workers. In bi-partition, memory cost in the FP owner is negligible, significantly reducing the former worker's memory overhead by putting the FP on the former worker and BP on the latter worker. Thus, bi-partition makes full use of the memory capability of the pipeline.

Implementation

We implemented a Python library that uses nearly 10000 LOC to manage the worker nodes, model deployment, and handle communication.

Worker Management

The master node(coordinator) is responsible for managing the task assignment of each worker, log system, and managing training and testing data and aggregating analysis results. After the coordinator starts running, each worker can access and register to the coordinator. The coordinator records the IP address of each worker, and the device status (storage capacity, computing capacity, and network card bandwidth), and assigns a unique ID value for each worker. The log information includes 4 levels (INFO, WARN, ERROR, DEBUG), which mainly records the operation status of the system during training, the size of packets sent and received by each worker, and the executing time.

Task Types

The training tasks assigned to each worker are divided into 3 categories. 1) Forward and backward: this part of the task is consistent with the normal worker task after model partitioning, sending the forward computation results to the next node and the backward computation results to the previous node. 2) Forward only: this task, after receiving the input data, performs the computation (this worker does not need to store the activation), while the task has a part of the task to update the model and receive the model update result of the backward pass to update its own model parameters for the next forward pass task. 3) Backward only: the calculation task is similar to 2), except that when executing the backward pass, sending the updated model parameters also need to update their own model parameters simultaneously, while accepting the activation value of the corresponding layer in 2) as the input for the next backward pass.

Task Assignment

After all the workers are registered to the coordinator, the target DNN model is simply partitioned (within the storage limit of a single machine) into several workers for profiling. The training data is sent to the first worker with a few epochs, and the execution time, activation size, and gradient size of each layer are recorded. When starting to partiton the DNN network using the bi-partition algorithm, the coordinator will use the current saved information of each worker as the constraint to finally arrive at the optimal partition result. When implementing the 1F1B strategy, each worker will cycle through their own task, blocking the data reading of the previous worker's data and executing the subsequent computation tasks when the reception is completed. In this way, only the coordinator needs to continuously loop the input data into the first worker node to control the 1F1B parallel training of the whole pipeline. It is worth noting that there will be some differences in the training tasks in the startup phase and the tasks in the steady phase, which need to be handled especially according to the number of workers currently joining the pipeline.

Communication Handel

RPC and SOCKET communication methods are provided. SOCKET communication method is more intuitive to understand, for each worker, it only needs to save the SOCKET_ADDR of the previous node and the next node, establish TCP connection through accept, and then it can transmit data in a similar way as reading and writing files, which also needs to be implemented through the serialization interface to ensure that the object data transmission process can be continued through deserialization after the rest of the computational tasks.

Performance evaluation

Experimental setup

Task and Datasets

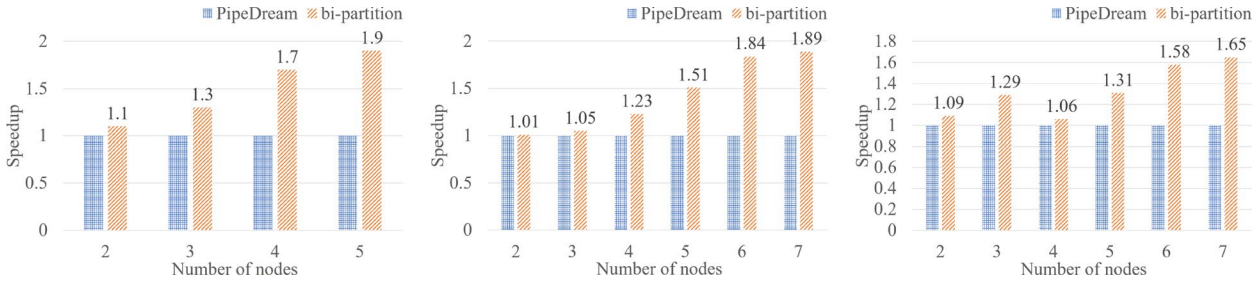
We consider the typical DNN task, i.e., image classification. We use two datasets to train the DNN models. 1) MNIST, a well-known handwritten digit recognition dataset. 2) CIFAR10, a subset of the 80 million tiny images dataset.

Devices

We use virtual machine technology to build 7 devices for our experiment. Devices are homogeneous with a 1-core, 2-thread processor, 2GB of RAM, 30GB of disk, and 100Mbps bandwidth bridged to the physical network. All the devices are running 64-bit Ubuntu 20.04.1 server with python3.8 and TensorFlow [25]2.5.0.

Models

We use three typical DNN models in our experiments: 1) LeNet-5 [26] with 60,000 parameters, 2) VGG16 [27] with 130 million parameters, and 3) AlexNet [28] with 61 million parameters. These three classical CNN models have the common characteristics of almost mainstream CNN models, and their stability has been tested over time, which can well represent the current CNN network models. The three representative network models selected in our experiment can be used to test the universality of the bi-partition algorithm. Because of the common characteristics of neural networks, both forward and backward propogations are required. On larger-scale DNN models like ResNet [29] and Transformer [30], our bi-partition algorithm is also effective. However, considering the rapid development of DNN, the size of DNN model parameters is also growing rapidly. It is not common to train large models such as GPT-3 [31] or Bert [32], but it is common to train DNN with poor-performance machines. It is still of practical significance



(a) LeNet-5 MNIST. (b) AlexNet CIFAR10. (c) VGG16 CIFAR10.

Fig. 8 Speedup on LeNet-5, AlexNet and VGG16 compared to PipeDream

to train these three representative networks using these devices that can not train VGG-16 by a single. Through some simple extensions, the bi-partition algorithm can also be directly extended to large-scale networks.

Training Methods

We use three different training methods to train the three typical DNN models, respectively. The optimizer is Adam (alpha=0.00003) and the training epochs of 20. The rest setups for hyper-parameters are as follows, 1) LeNet-5, we use per-device batch size=64 to train on MNIST dataset. 2) VGG16, we use per-device batch size=128 to train on CIFAR10 dataset. 3) AlexNet, we use per-device batch size=64 to train on CIFAR10 dataset.

Evaluation

Speedup

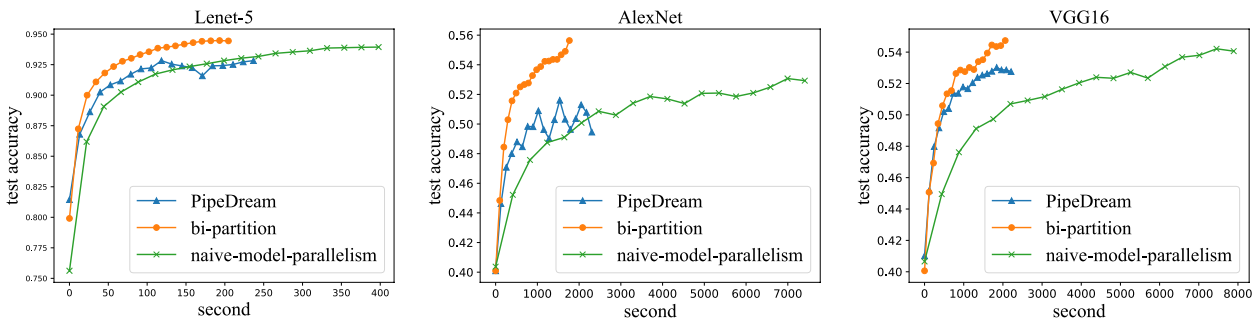
The speedup ratio is the ratio of the time T_p taken by PipeDream [20] to the time T_b taken by bi-partition for the same number of training epochs. Therefore the speedup ratio of PipeDream is always 1, and the bi-partition is T_p/T_b. The purpose of the first experiment is to compare the training efficiency of the traditional split-by-layer (PipeDream) and our approach. Figure 8 shows the experimental results on three groups a), b), and c) of different models and datasets, respectively. Each of the groups uses the same experimental setup except for the number of devices.

In Fig. 8, the horizontal coordinate represents the number of devices, and the vertical coordinate shows the training speedup ratio. It is obvious that the training efficiency of bi-partition outperforms that of PipeDream for various DNNs and datasets. Figure 8a shows 1.9× speedup ratio with 5 nodes. In each group, we find that the improvement in training efficiency becomes more significant as the number of devices increases. This is due to the fact that the bi-partition approach is more fine-grained than PipeDream, which causes PipeDream to encounter bottlenecks(the training efficiency of the entire pipeline is limited by the execution time of one layer) earlier as the number of devices increases. This trend indicates that the bi-partition has better system scalability than PipeDream.

Test Accuracy & Training Loss

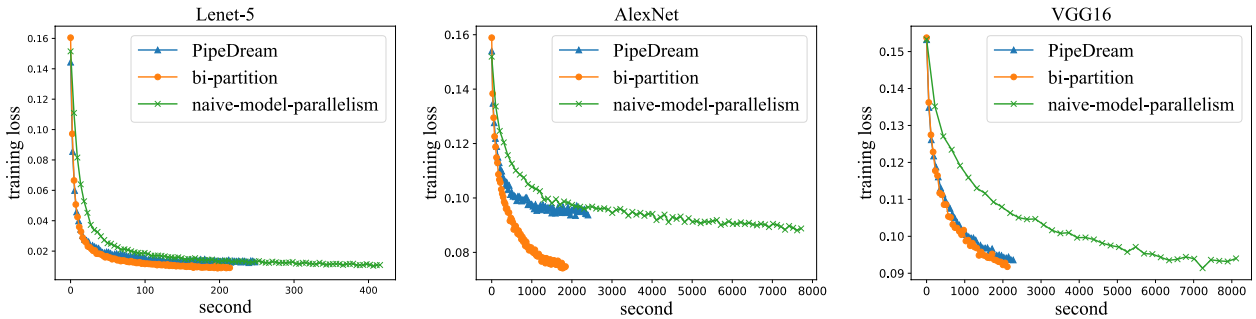
The purpose of the second experiment is to investigate the training convergence performance of PipeDream and bi-partition. Figures 9 and 10 show three groups experiments under different settings, as the sub-label showing.

It can be seen that bi-partition is faster than the PipeDream to finish 20 epochs tasks because of its ability to fully utilize the computational resources in the pipeline.



(a) LeNet-5 MNIST $m = 2$. (b) AlexNet CIFAR10 $m = 4$. (c) VGG16 CIFAR10 $m = 4$.

Fig. 9 Test accuracy vs. Training time on LeNet-5, AlexNet and VGG16 compared to naive-model-parallelism and PipeDream



(a) LeNet-5 MNIST $m = 2$. (b) AlexNet CIFAR10 $m = 4$. (c) VGG16 CIFAR10 $m = 4$.

Fig. 10 Training Loss vs. Training time on LeNet-5, AlexNet and VGG16 compared to naive-model-parallelism and PipeDream

Moreover, due to the finer-grained partitioning of bi-partition, some devices have only forward or backward computation, while some devices have only backward computation, which as reduces of the number of stages. In 1F1B schedule, the number of stages increasing the number of copies to be stored, which also means less gradient staleness. Figure 9b demonstrates this phenomenon. The result of the bi-partition partition allows one device to only perform the backward computation, so there are actually only three stages instead of the four stages of the PipeDream. The experimental results also prove that Bi-partition obtains a better convergence performance than PipeDream.

Conclusion

In the model partition stage of PMP, to tackle the problem of low computational efficiency of large scale DNN training caused by uneven model partition, we propose a new bi-partition method that considering the differences of computational pattern between FP and BP. The core idea is to separately partition the DNN model for the FP and BP. Based on dynamic programming, we design an efficient algorithm to find the optimal model partition result for minimizing the pipeline parallelism training time. Meanwhile, the proposed method greatly improves the resource utilization for DNN training in the cloud. In our experiment, compared to state-of-the-art PipeDream, the training efficiency of bi-partition is up to 1.9× faster on various typical DNN models and the number of devices.

Meanwhile, our proposed model partitioning algorithm is not only applicable to the popular DNN architecture like convolutional neural networks (CNN), but also applicable to recurrent neural network (RNN) and Transformer architecture by making slight modifications. As for other networks like RNN and Transformer, the training processes including BP and FP are similar to CNN. However, since the computational complexity per layer in RNN or per block in Transformer is similar, it is important to consider the scenarios where the computational power and communication bandwidth of the devices are heterogenous. These scenarios make the problem more complicated and will be considered in our future work.

Authors' contributions
Lingyun Cui is mainly responsible for the idea of this manuscript and the design and implementation of the system and algorithm. Zhihao Qu gave many suggestions on system design and algorithm correctness. In addition, he also helped revise the article. Guomin Zhang provided suggestions for revising this manuscript. Bin Tang put forward many constructive suggestions on the motivation and significance of this article. Baoliu Ye provided the necessary environment and equipment for the experiment, as well as academic guidance for this article. The author(s) read and approved the final manuscript.

Funding
This research was supported by the Fundamental Research Funds for the Central Universities under Grant No. B210202079 and B210201053, the National Natural Science Foundation of China under Grant No.62102131, No.61832005 and No.61872171, the Natural Science Foundation of Jiangsu Province under Grant BK20210361 and BK20190058, the Water Conservancy Science and Technology Project of Jiangsu Province under Grant No. 2021053, and the Future Network Scientific Research Fund Project under Grant No. FNSRFP-2021-ZD-07.

Availability of data and materials
Data are available on the websites: MNIST: http://yann.lecun.com/exdb/mnist/; CIFAR10: https://www.cs.toronto.edu/~kriz/cifar.html. The code has been open-sourced to https://github.com/cccccly/Parallel-SGD.

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
Hohai University, Army Engineering University and Nanjing University.

Received: 14 October 2022 Accepted: 20 December 2022
Published online: 16 February 2023

References

1. Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai X, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S, et al (2021) An image is worth 16x16 words: Transformers for image recognition at scale. In: Proc of the ICLR. OpenReview.net, Austria
2. Liu Z, Lin Y, Cao Y, Hu H, Wei Y, Zhang Z, Lin S, Guo B (2021) Swin transformer: Hierarchical vision transformer using shifted windows. In: Proc. of the IEEE/CVF ICCV. IEEE, Montreal, BC, Canada, p 10012–10022
3. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Proc of the NeurIPS, vol 30. Curran Associates Inc.57, Long Beach, CA, USA
4. Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K, et al (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. preprint ArXiv:1609.08144
5. Shoeybi M, Patwary M, Puri R, LeGresley P, Casper J, Catanzaro B (2019) Megatron-LM: Training multi-billion parameter language models using model parallelism. preprint ArXiv:1909.08053
6. Fedus W, Zoph B, Shazeer N (2021) Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. preprint ArXiv:2101.03961
7. Huang Y, Xu H, Gao H, Ma X, Hussain W (2021) Ssur: An approach to optimizing virtual machine allocation strategy based on user requirements for cloud data center. IEEE Trans Green Commun Netw 5(2):670–681. https://doi.org/10.1109/TGCN.2021.3067374
8. Mohamed I, Al-Mahdi H, Tahoun M, Nassar H (2022) Characterization of task response time in fog enabled networks using queueing theory under different virtualization modes. J Cloud Comput 11(1):1–17
9. Luo Q, Hu S, Li C, Li G, Shi W (2021) Resource scheduling in edge computing: A survey. IEEE Commun Surv Tutorials 23(4):2131–2165. https://doi.org/10.1109/COMST.2021.3106401
10. Pang M, Wang L, Fang N (2020) A collaborative scheduling strategy for iov computing resources considering location privacy protection in mobile edge computing environment. J Cloud Comput 9(1):1–17
11. Li M, Andersen DG, Park JW, Smola AJ, Ahmed A, Josifovski V, Long J, Shekita EJ, Su BY (2014) Scaling distributed machine learning with the parameter server. Proc OSDI 14:583–598. USENIX Association, Broomfield, CO, USA
12. Gupta V, Choudhary D, Tang P, Wei X, Wang X, Huang Y, Kejariwal A, Ramchandran K, Mahoney MW (2021) Training recommender systems at scale: Communication-efficient model and data parallelism. In: Proc. of the SIGKDD. ACM, Singapore, p 2928–2936
13. Rothchild D, Panda A, Ullah E, Ivkin N, Stoica I, Braverman V, Gonzalez J, Arora R (2020) FetchSGD: Communication-efficient federated learning with sketching. In: Proc. of the ICML. PMLR. Vienna, Austria, p 8253–8265
14. Chen CY, Ni J, Lu S, Cui X, Chen PY, Sun X, Wang N, Venkataramani S, Srinivasan VV, Zhang W et al (2020) Scalecom: Scalable sparsified gradient compression for communication-efficient distributed training. Proc NeurIPS 33:13551–13563
15. Guo S, Qu Z (2022) Edge Learning for Distributed Big Data Analytics: Theory, Algorithms, and System Design. Cambridge University Press, United Kingdom
16. Zhang J, Qu Z, Chen C, Wang H, Zhan Y, Ye B, Guo S (2021) Edge learning: The enabling technology for distributed big data analytics in the edge. ACM Comput Surv (CSUR) 54(7):1–36
17. Qu Z, Guo S, Wang H, Ye B, Wang Y, Zomaya A, Tang B (2021) Partial synchronization to accelerate federated learning over relay-assisted edge networks. IEEE Trans Mobile Comput 21(12):4502–4516
18. Huang Y, Cheng Y, Bapna A, Firat O, Chen D, Chen M, Lee H, Ngiam J, Le QV, Wu Y, et al (2019) Gpipe: Efficient training of giant neural networks using pipeline parallelism. In: Proc of the NeurIPS, vol 32. Curran Associates Inc.57, Vancouver, BC, Canada
19. Li S, Hoefler T (2021) Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In: Proc. of the SC. ACM, St. Louis, Missouri, USA, p 1–14
20. Narayanan D, Harlap A, Phanishayee A, Seshadri V, Devanur NR, Ganger GR, Gibbons PB, Zaharia M (2019) PipeDream: generalized pipeline parallelism for DNN training. In: Proc. of the SOSP. ACM, Huntsville, ON, Canada, p 1–15
21. Narayanan D, Phanishayee A, Shi K, Chen X, Zaharia M (2021) Memory-efficient pipeline-parallel dnn training. In: Proc of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event, p 7937–7947
22. Fan S, Rong Y, Meng C, Cao Z, Wang S, Zheng Z, Wu C, Long G, Yang J, Xia L, et al (2021) DAPPLE: A pipelined data parallel approach for training large models. In: Proc. of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, p 431–445
23. Park JH, Yun G, Chang MY, Nguyen NT, Lee S, Choi J, Noh SH, Choi Yr (2020) HetPipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, p 307–321
24. Zhao S, Li F, Chen X, Guan X, Jiang J, Huang D, Qing Y, Wang S, Wang P, Zhang G et al (2021) v pipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training. IEEE Trans Parallel Distrib Syst 33(3):489–506
25. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al (2016) TensorFlow: a system for Large-Scale machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16). USENIX Association, GA, USA, p 265–283
26. LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324
27. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. preprint ArXiv:1409.1556
28. Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Proc of the NeurIPS, vol 25. Curran Associates Inc.57, Lake Tahoe, Nevada, USA
29. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proc. of the IEEE conference on computer vision and pattern recognition. pp 770–778
30. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. Advances in neural information processing systems, vol 30. Curran Associates Inc.57, Long Beach, CA, USA
31. Floridi L, Chiriatti M (2020) Gpt-3: Its nature, scope, limits, and consequences. Minds Mach 30(4):681–694
32. Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805

Publisher's Note