Journal of Parallel and Distributed Computing 193 (2024) 104947

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

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

PiPar: Pipeline parallelism for collaborative machine learning

Zihan Zhang [a,*], Philip Rodgers [b], Peter Kilpatrick [c], Ivor Spence [c], Blesson Varghese [a]

[a] *School of Computer Science, University of St Andrews, St Andrews, United Kingdom*
[b] *Rakuten Mobile Inc., Tokyo, Japan*
[c] *School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast, United Kingdom*

ARTICLE INFO

Keywords:
Collaborative machine learning
Resource utilization
Pipeline parallelism
Edge computing

ABSTRACT

Collaborative machine learning (CML) techniques, such as federated learning, have been proposed to train deep learning models across multiple mobile devices and a server. CML techniques are privacy-preserving as a local model that is trained on each device instead of the raw data from the device is shared with the server. However, CML training is inefficient due to low resource utilization. We identify idling resources on the server and devices due to sequential computation and communication as the principal cause of low resource utilization. A novel framework PiPar that leverages pipeline parallelism for CML techniques is developed to substantially improve resource utilization. A new training pipeline is designed to parallelize the computations on different hardware resources and communication on different bandwidth resources, thereby accelerating the training process in CML. A low overhead automated parameter selection method is proposed to optimize the pipeline, maximizing the utilization of available resources. The experimental results confirm the validity of the underlying approach of PiPar and highlight that when compared to federated learning: (i) the idle time of the server can be reduced by up to 64.1×, and (ii) the overall training time can be accelerated by up to 34.6× under varying network conditions for a collection of six small and large popular deep neural networks and four datasets without sacrificing accuracy. It is also experimentally demonstrated that PiPar achieves performance benefits when incorporating differential privacy methods and operating in environments with heterogeneous devices and changing bandwidths.

1. Introduction

Deep learning has found application across a range of fields including computer vision [17,12], natural language processing [5,2] and speech recognition [7,10]. However, there are important data privacy and regulatory concerns in sending data generated on mobile devices to geographically distant cloud servers for training deep learning models. A new class of machine learning techniques has therefore been developed under the umbrella of collaborative machine learning (CML) to mitigate these concerns [38]. CML does not require data to be sent to a server for training deep learning models; rather the server shares models with devices that are then locally trained on the device.

CML is used in many real-world use-cases comprising a central server and multiple homogeneous mobile devices. Smartphone manufacturers, for example, analyse user data to improve the performance of a specific smartphone model [44,47]. For instance, CML can be employed to analyse the battery usage patterns of individual users on their phones to offer personalized plans for optimizing battery life. Similarly, CML can be used to analyze the typing habits of the users and then automatically complete and correct the typing of the users.

There are three notable CML techniques reported in the literature, namely federated learning (FL) [19–21,28], split learning (SL) [8,39] and split federated learning (SFL) [37,41]. However, these techniques under-utilize both compute and network resources, which results in training times that do not meet real-world requirements. The cause of resource under-utilization and the resulting performance inefficiency in the three CML techniques is considered next.

In FL, each device trains a local model of a deep neural network (DNN) using the data it generates. Local models are uploaded to the server and aggregated as a global model at a pre-defined frequency. However, the workload of the devices and the server is usually imbalanced [16,46,41]. This is because the server is only employed when the local models are aggregated and is idle for the rest of the time.

In SL, a DNN is usually decomposed into two parts, such that the initial layers of the DNN are deployed on a device and the remaining layers on the server. A device trains the partial model and sends the intermediate outputs to the server where the rest of the model is trained.

* Corresponding author.
E-mail address: zz66@st-andrews.ac.uk (Z. Zhang).

https://doi.org/10.1016/j.jpdc.2024.104947
Received 9 January 2024; Received in revised form 10 April 2024; Accepted 21 June 2024
Available online 2 July 2024

The training of the model on devices occurs in a round-robin fashion. Hence, only one device or the server will utilize its resources while the rest of the devices or the server are idle [37,38].

In SFL, which is a hybrid of FL and SL, the DNN is split across devices and the server. The devices, however, unlike SL, train the local models concurrently. Nevertheless, the server must wait while the devices train the model and transfer data, and vice versa.

Therefore, the following two challenges need to be addressed for improving resource utilization in CML:

a) Sequential execution on devices and the server causes resource under-utilization: For FL, the server aggregates the models obtained from all devices after they complete training; for SL, after the training of the initial layers is completed on the devices, the remaining layers of the DNN are trained on the server. Since device-side and server-side computations in CML techniques occur in sequence, there are long idle times on both the devices and the server.

b) Communication between devices and the server results in resource under-utilization: Data transfer in CML techniques is time consuming [3, 36,6], during which time no training occurs on both the server and devices. This increases the overall training time.

Although low resource utilization of CML techniques makes training inefficient, there is currently limited research that is directed at addressing this problem. This paper aims to address the above challenges by developing a framework, PiPar (pronounced as 'piper'), that leverages *pipeline parallelism* to improve the resource utilization of devices and servers in CML techniques when training DNNs, thereby increasing training efficiency. The framework distributes the computation of DNN layers on the server and devices, balances the workload on both the server and devices and reorders the computation for different inputs in the training process. PiPar overlaps the device and server-side computations with communication between the devices and server, thereby improving resource utilization, which in turn accelerates CML training.

PiPar redesigns the training process of DNNs. Traditionally, training a DNN involves the forward propagation pass (or forward pass) and backward propagation pass (or backward pass). In the forward pass, one batch of input data (also known as a mini-batch) is used as input for the first DNN layer and the output of each layer is passed on to subsequent layers to compute the loss function. In the backward pass, the loss function is passed layer by layer from the last layer to the first layer to compute the gradients of the DNN model parameters.

PiPar divides the DNN into two parts and deploys them on the server and devices as in SFL. Then the forward and backward passes are reordered for multiple mini-batches to reduce idle time. Each device executes the forward pass for multiple mini-batches in sequence. The immediate result of each forward pass (activations) is transmitted to the server, which executes the forward and backward passes for the remaining layers and transfers the gradients of the activations back to the device. The device then sequentially performs the backward passes for the mini-batches. The devices operate in parallel, and the local models are aggregated at a set frequency. Since many forward passes occur sequentially on the device, the communication for each forward pass overlaps the computation of the subsequent forward passes. Also, in PiPar, the server and device computations occur simultaneously for different mini-batches. Thus, PiPar reduces the idle time of devices and servers by overlapping server and device-side computations and server-device communication.

This paper makes the following contributions:

(1) The development of a novel framework PiPar to accelerate collaborative training of DNNs by improving resource utilization. PiPar is the first work to reduce resource idling in CML by reordering training tasks across a server and the participating devices. Idle time is reduced by leveraging pipeline parallelism to overlap device and server computations and device-server communication.

(2) Development of an low overhead automated parameter selection approach for further optimizing CML workloads across devices and servers to maximize the overall training efficiency.

PiPar and the automated parameter selection approach are evaluated on a lab-based testbed. The experimental results demonstrate that: a) compared to FL, PiPar can accelerate the training process by up to 34.6 ×, and the idle time of hardware resources is reduced by up to 64.1 ×. b) the automated parameter selection approach can find the optimal or near-optimal parameters in less time than an exhaustive search. It is also experimentally demonstrated that PiPar achieves performance benefits when incorporating differential privacy methods and operating in environments with heterogeneous devices and changing bandwidths.

The rest of this paper is organized as follows. Section 2 considers the background and work related to this research. The PiPar framework and the two approaches that underpin the framework are detailed in Section 3. Section 4 provides a theoretical analysis of model convergence and accuracy using PiPar. Experiments in Section 5 demonstrate the effectiveness of the PiPar framework. Section 6 concludes this article.

2. Background and related work

Section 2.1 provides the background of collaborative machine learning (CML), and Section 2.2 introduces the related research on improving the training efficiency in CML.

2.1. Background

The training process of three popular CML techniques, namely federated learning (FL), split learning (SL) and split federated learning (SFL), and their limitation due to resource under-utilization are presented.

2.1.1. Federated learning

FL [19–21,28] uses a set of devices coordinated by a central server to train deep learning models collaboratively.

Assume K devices participate in the training process as shown in Fig. 1(a). In Step ①, the devices train the complete model M^k locally, where $k = 1, 2, ..., K$. In each iteration, the local model trains on a mini-batch of data by completing the forward and backward passes to compute gradients of all model parameters and then update the parameters with the gradients. A training epoch involves training over the entire dataset, which consists of multiple iterations. In Step ②, after a predefined number of local epochs, the devices send the local models M^k to the server, where $k = 1, 2, ..., K$. In Step ③, the server aggregates the local models to obtain a global model M, using the FedAvg algorithm [28]; $M = \sum_k \frac{|\mathcal{D}^k|}{\sum_k |\mathcal{D}^k|} M^k$, where $\mathcal{D}_k$ is the local dataset on device k and $|\cdot|$ is the function to obtain the set size. In Step ④, the global model is downloaded to the devices. The next round of training continues until the model converges.

Typically, local model training on devices (Step ①) takes most of the time, while the server with more significant compute performance is idle. Therefore, PiPar utilizes the idle resources on the server during training.

2.1.2. Split learning

SL [8,39] is another privacy-preserving CML method. Since a DNN consists of consecutive layers, SL splits the entire DNN M into two parts at the granularity of layers and deploys them on the server (M^s) and the devices (M^{c_k}, where $k = 1, 2, ..., K$).

As shown in Fig. 1(b), the devices train the initial layers of the DNN and the server trains the remaining layers, and the devices work in a round-robin fashion. In Step ①, the first device executes the forward pass of M^{c_1} on its local data, and in Step ②, the intermediate results, also known as activations, are sent to the server. In Step ③, the server uses the activations to complete the forward pass of M^s to obtain the loss. The loss is then used for the backward pass on the server to compute the gradients of the parameters of M^s and the gradients of the activations. In Step ④, the gradients of the activations are sent back to the device,

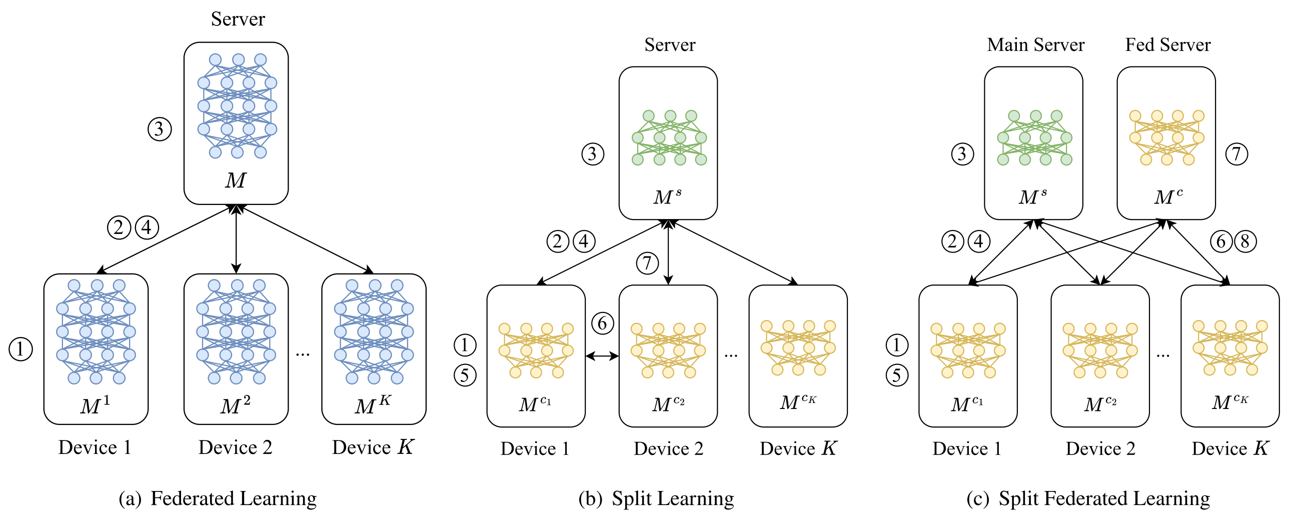


Fig. 1. Training of CML methods, assuming K devices. The training steps (circled numbers) are explained in Section 2.1.

and in Step ⑤, the gradients of the parameters of M^{c_1} are computed in the device-side backward pass. Next, the parameters of the server-side model and device-side model are updated by their gradients. In Step ⑥, after a device trains for a certain number of epochs, the next device gets the latest model from the previous device and starts training its model in Step ⑦.

Compared to FL, device-side computation is significantly reduced because only a few layers are trained on devices. However, since the devices work in sequence (instead of in parallel as FL), the overall training efficiency decreases as the number of devices increases.

2.1.3. Split federated learning

Since FL is computationally intensive on devices and SL works inefficiently on the device, SFL [37] was developed to alleviate both limitations. Similar to SL, SFL also splits the DNN across the devices (M^{c_k}, where $k = 1, 2, ..., K$) and the server (M^s) and collaboratively trains the DNN. However, in SFL, the devices train in parallel and utilize a 'main' server for training the server-side model and a 'Fed' server for aggregation.

The training process is shown in Fig. 1(c). In Step ①, the forward pass of M^{c_k}, where $k = 1, 2, ..., K$, are executed on the devices in parallel, and in Step ②, the activations are uploaded to the main server. In Step ③, the main server trains M^s, and in Step ④, the gradients are sent back to all devices before they complete the backward pass in Step ⑤. At a predefined frequency, the models M^{c_k}, where $k = 1, 2, ..., K$, are uploaded to the Fed server in Step ⑥. In Step ⑦, the models are aggregated to the global model M^c. In Step ⑧, M^c is downloaded to the devices and used for the next round of training.

SFL utilizes device parallelism to improve the training efficiency of SL [38]. However, the server still waits while the devices are training the model (Step ①) and transmitting data (Step ②), and vice versa, which leads to resource under-utilization. PiPar addresses this problem by parallelizing the steps performed on the server and the devices.

2.2. Related work

Existing research to improve the training efficiency of CML focuses on the three aspects considered below.

2.2.1. Improving resource utilization using pipeline parallelism

Approaches employing pipeline parallelism have been proposed to improve the compute and network utilization of resources. GPipe [15] and PipeDream [29] use pipeline parallelism when a DNN is distributed to multiple computing nodes and parallelizes the computations on different nodes. They both reduce the idle time on computing resources. To further improve the hardware utilization, PipeMare [45] implements asynchronous training, and Chimera [25] uses bidirectional pipelines instead of a unidirectional one. PipeFisher [31] takes advantage of idle resources to execute second-order optimization to accelerate model convergence. However, these approaches for distributed DNN training cannot be directly applied to CML for three reasons.

Firstly, the context in which current pipeline parallelism approaches were designed to operate in is completely different from CML. They were designed for GPU clusters with substantial compute resources where the data flow is sequential (data is provided as input to one node and then the output goes to the next node and so on). However, in CML, the data generated on end-user devices is not shared with other devices or servers to preserve privacy. PiPar is therefore proposed to tackle the problem of distributed training of devices in centralized topologies.

Secondly, in existing pipeline parallelism approaches, different layers of the DNN are mapped onto different nodes in a cluster and they do not share weights. However, in CML, each device trains a local model on its data, and the model weights are subsequently synchronized with other devices. PiPar splits the model across the server and all the devices to alleviate the computational burden on the devices and proposes a method to synchronize the server-side and client-side models.

Thirdly, the bandwidth between different devices and servers in CML will be variable as seen in real-world mobile environments. However, the bandwidth between the nodes of a GPU cluster is relatively less prone to such variability. Since communication time between nodes of a GPU cluster is relatively small, existing pipeline parallelism approaches tend to hide communication behind computation. However, the communication of activations and gradients in CML is a substantial volume, which is not handled by existing approaches. PiPar takes this into account, and hence, a parameter selection method is proposed to overlap the communication and computation.

Given the above limitations, we propose a novel framework, PiPar that fully utilizes the computing resources on the server and devices and the bandwidth available between them to improve the training efficiency of CML.

2.2.2. Reducing the impact of stragglers

Stragglers among the devices used for training increase the overall training time of CML. A device selection method was proposed based on the resource availability of devices to minimize the impact of stragglers [30]. Certain neurons of the DNN on a straggler are masked to accelerate computation [43]. Local gradients were aggregated hierarchically to accelerate FL on heterogeneous devices [40]. To balance workloads across heterogeneous devices, FedAdapt [41] offloaded DNN layers from devices to a server. An adaptive asynchronous federated learning mechanism [26] was proposed to mitigate stragglers.

These methods alleviated the impact of stragglers but did not address the fundamental challenge of sequential computation and communication between the devices and server that results in low resource utilization.

2.2.3. Reducing communication overhead

In limited bandwidth environments, communication overhead limits the training efficiency of CML techniques. To reduce the communication traffic in FL, a relay-assisted two-tier network was developed [33]. Models and gradients were transmitted simultaneously and aggregated on the relay nodes. Pruning, quantization and selective updating were used to reduce the model size and thus reduce the computation and communication overhead [42]. The communication involved in the backward pass of SFL was improved by averaging the gradients on the server-side model and broadcasting them to the devices instead of unicasting the unique gradients to devices [32]. Overlap-FedAvg [48] was proposed to decouple the computation and communication during training and overlap them to reduce idle resources. However, the use of computing resources located at the server was not fully leveraged.

These methods are effective in reducing the data volume transferred over the network, thus reducing the communication overhead. However, this reduces model accuracy.

3. PiPar

This section develops PiPar, a framework to improve the resource utilization of CML in the context of FL and SFL. PiPar accelerates the execution of sequential DNNs for the first time by leveraging pipeline parallelism to improve the overall resource utilization in centralized CML.

The PiPar framework is underpinned by two approaches, namely *pipeline construction* and *automated parameter selection*. The first approach constructs a training pipeline to balance the overall training workload by (a) reallocating the computations for different DNN layers on the server and devices, and (b) reordering the forward and backward passes for multiple mini-batches of data by scheduling them onto idle resources. Consequently, not only is the resource utilization improved by using PiPar, but also the overall training of the DNN. The second approach of PiPar enhances the performance of the first approach by automatically selecting the optimal control parameters (such as the point at which the DNN must be split across the device and the server and the number of mini-batches that can be executed concurrently in the pipeline).

3.1. Motivation

The following three observations on low resource utilization when training DNNs in CML motivate PiPar.

(1) The server and devices need to work simultaneously: The devices and server work in an alternating manner in the current CML methods, which is a limitation that must be addressed to improve resource utilization. In FL, the server starts to aggregate local models only after all devices have completed training their local models. In SL/SFL, the sequential computation of DNN layers results in the sequential working of the devices and the server. The dependencies between server-side and device-side computations need to be eliminated to reduce the resulting idle time on the resources. PiPar attempts to make the server and the devices work simultaneously by reallocating and reordering training tasks.

(2) Compute-intensive and I/O-intensive tasks need to be overlapped: Compute-intensive tasks, such as model training, involve large-scale computations performed by computing units (CPU/GPU), while IO-intensive tasks refer to input and output tasks of disk or network, such as data transmission, which usually do not have a high CPU requirement. A computationally intensive and an I/O-intensive task can be executed in parallel on the same resource without mutual dependencies. However, in current CML methods, both server-side and device-side computations are paused when communication is in progress, which creates idle time on compute resources. PiPar improves this by overlapping compute-intensive and I/O-intensive tasks.

(3) Workloads on the server-side and client-side need to be balanced: Idle time on resources is also caused due to imbalanced workloads on the server and devices. PiPar balances the workloads on the server and device sides by splitting the DNN carefully.

3.2. Pipeline construction

Assume that K devices and a server train a sequential DNN collaboratively by using data residing on each device. Conventionally, the dataset on each device is divided into multiple mini-batches that are fed to the DNN in sequence. Training on each mini-batch involves a forward pass that computes a loss function and a backward pass that computes the gradients of the model parameters. A training epoch ends after the entire dataset has been fed to the DNN. To solve the problem of low resource utilization faced by the current CML methods, PiPar constructs a training pipeline that reduces the idle time on resources during collaborative training.

Each forward and backward pass of CML methods comprises four tasks: (i) the device-side compute-intensive tasks, such as model training; (ii) the device-to-server I/O-intensive task, such as data uploading; (iii) the server-side compute-intensive task, such as model training (only in SL and SFL) and model aggregation; (iv) the server-to-device I/O-intensive task, such as data downloading. The four tasks can only be executed in sequence in current CML methods, resulting in idle resources. To solve this problem, a pipeline is developed to balance and parallelize the above tasks. The pipeline construction approach involves three phases, namely DNN splitting, training stage reordering and multi-device parallelization.

3.2.1. Phase 1 - DNN splitting

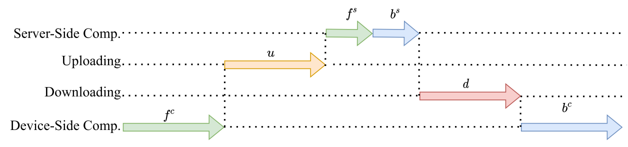
The approach aims to overlap the above-mentioned four tasks to reduce idle time on computing resources on the server and devices as well as idle network resources. Since this approach does not reduce the computation and communication time of each task, it needs to balance the time required by the four tasks to avoid straggler tasks from increasing the overall training time. For example, in FL, the device-side compute-intensive task is the most time-consuming, while the other three tasks require relatively less time. In this case, overlapping the four tasks will not significantly reduce the overall training time. Therefore, it is more appropriate to split the DNN and divide the training task across the server and the devices (similar to previous works [37,41]). In addition, since the output of each DNN layer has a variable size, different split points of the DNN will result in different volumes of transmitted data. Thus, changing the split point based on the computing resources and bandwidth can also balance the I/O-intensive tasks with compute-intensive tasks. The selection of the best splitting point is presented in Section 3.3.

Splitting DNNs does not affect model accuracy, since it does not alter computations but rather the resource on which they are executed. In FL, each device k, where $k = 1, 2, ..., K$, trains a complete model M^k. PiPar splits M^k to a device-side model M^{c_k} and a server-side model M^{s_k} represented as:

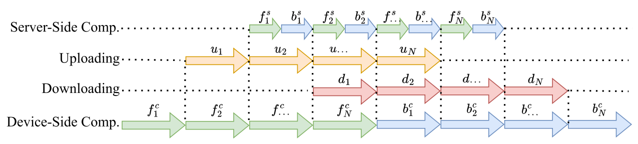
$$M^k = M^{s_k} \oplus M^{c_k} \tag{1}$$

where the binary operator $\oplus$ stacks the layers of two partitions of a DNN as a complete DNN.

There are k pairs of $\{M^{c_k}, M^{s_k}\}$, where M^{c_k} is deployed on device k while all of M^{s_k} are deployed on the server. This is different from SL and SFL where only one model is deployed on the server-side. Assume



(a) Conventional training pipeline when using a split DNN


Server-Side Comp.
Uploading
Downloading
Device-Side Comp.
u_1 u_2 $u_{...}$ u_N
d_1 d_2 $d_{...}$ d_N
f_1^c f_2^c $f_{...}^c$ f_N^c b_1^c b_2^c $b_{...}^c$ b_N^c

(b) Training pipeline in PiPar. N mini-batches are trained in parallel in each training iteration; subscripts indicate the index of the mini-batch.

Fig. 2. Pipelines for one training iteration in conventional training and PiPar when using a split DNN. "Comp" is an abbreviation for computation. f, b, u and d represent forward pass, backward pass, upload and download, respectively. Superscripts indicate server-side (s) or client-side (c) computation or communication.

the complete model M^k contains Q layers, M^{c_k} contains the initial P layers and M^{s_k} contains the remaining layers, where $1 \leq P \leq Q$.

Splitting the DNN maintains the consistency of the training process and does not change the model accuracy; this is demonstrated in Section 4.1.

3.2.2. Phase 2 - training stage reordering

After splitting the DNNs and balancing the four tasks, idle resources in the training process need to be utilized. This is achieved by reordering the computations for different mini-batches of data.

Fig. 2(a) shows the pipeline of one training iteration of a split DNN for one pair of $\{M^{s_k}, M^{c_k}\}$ (the device index k is not shown). Any forward pass (f), backward pass (b), upload task (u) and download task (d) for each mini-batch is called a *training stage*.

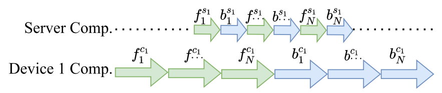
The idle time on the device exists between the forward pass f^c and the backward pass b^c of the device-side model. Thus, PiPar inserts the forward pass of the next few mini-batches into the device-side idle time to fill up the pipeline. As shown in Fig. 2(b), in each training iteration, the forward passes for N mini-batches, f_1^c to f_N^c, are performed on the device in sequence. The activations of each mini-batch are sent to the server (u_1 to u_N) once the corresponding forward pass is completed, which utilizes idle network resources. Once the activations of any mini-batch arrive, the server performs the forward and backward passes, (f_1^s, b_1^s) to (f_N^s, b_N^s), and sends the gradients of the activations back to the device (d_1 to d_N). After completing the forward passes of the mini-batches and receiving the gradients, the device performs the backward passes, b_1^c to b_N^c. Then the model parameters are updated and the training iteration ends. A training epoch ends when the entire dataset has been processed, which involves multiple training iterations.

Fig. 2(b) shows that compared to conventional training (Fig. 2(a)), the four tasks can be considerably overlapped and it is possible to significantly reduce the idle time of the server and the devices.

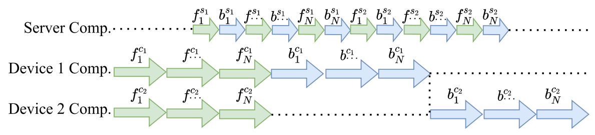
To guarantee a similar model accuracy as classic FL, the gradients must be obtained from the same number of data samples when the model is updated. This requires that the number of data samples involved in each training iteration in PiPar should be the same as the original batch size in FL. Since N mini-batches are used in each training iteration, the size of each mini-batch B' is reduced to $1/N$ of the original batch size B in FL.

$$B' = \lfloor B/N \rfloor \tag{2}$$

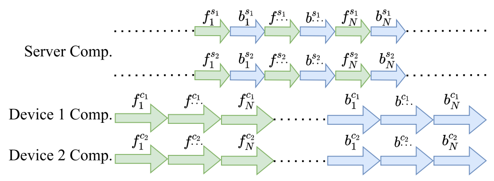
Reordering training stages does not impact model accuracy, which is demonstrated in Section 4.2.


Server Comp.
Device 1 Comp.
$f_1^{s_1}$ $b_1^{s_1}$ $f_{...}^{s_1}$ $b_{...}^{s_1}$ $f_N^{s_1}$ $b_N^{s_1}$
$f_1^{c_1}$ $f_{...}^{c_1}$ $f_N^{c_1}$ $b_1^{c_1}$ $b_{...}^{c_1}$ $b_N^{c_1}$

(a) PiPar with single device (same as Figure 2(b))



(b) PiPar with two devices in sequence



(c) PiPar with two devices in parallel

Fig. 3. PiPar using single and multiple devices. Comp, f, b, u and d represent computation, forward pass, backward pass, upload and download, respectively. The superscripts s_k and c_k represent the index of the model M^{s_k} and M^{c_k}, $k = 1, 2$, respectively.

3.2.3. Phase 3 - multi-device parallelization

The workloads of multiple devices involved in collaborative training need to be coordinated. On the device-side, each device k is responsible for training its model M^{c_k}, and PiPar allows them to train in parallel for efficiency. On the server-side, the counterpart K models (M^{s_1} to M^{s_K}) are deployed and trained simultaneously. However, this may result in contention for compute resources.

Fig. 3(a) shows the case of a single device (same as Fig. 2(b) but does not show communication), whereas Fig. 3(b) and Fig. 3(c) show the case of multiple devices. Fig. 3(b) offers a solution to train the server-side models sequentially. However, the server-side models that are trained relatively late will cause a delay in the backward passes for the corresponding device-side models, for example, $b_n^{c_2}$, where $n = 1, 2, ..., N$, in Fig. 3(b).

Alternatively, data parallelism can be employed. The activations from different devices are deemed as different inputs and the server-side models are trained in parallel on these inputs. This is shown in Fig. 3(c). It is worth noting that, compared to training a single model, training multiple models at the same time may result in longer training time for each model on a resource-limited server. This approach, nonetheless, mitigates stragglers on devices.

At the end of each training epoch, the device-side models M^{c_k} are uploaded to the server and will constitute the entire models M^k when combined with the server-side models M^{s_k} (Equation (1)). The complete model M^k of each device is aggregated to obtain a complete global model M, using the FedAvg algorithm [28].

$$M = \sum_{k=1}^{K} \frac{|\mathcal{D}^k|}{\sum_{k=1}^{K} |\mathcal{D}^k|} M^k \tag{3}$$

where $\mathcal{D}^k$ is the local dataset on device k and $|\cdot|$ is the function to obtain the size of the given dataset. The server-side global model M^s and device-side global model M^c are split from M using

$$M = M^s \oplus M^c \tag{4}$$

The devices download M^c to update the local models for the subsequent training epochs, and the server-side models are updated by M^s.

It has been shown in the previous phases that the accuracy of each local model M^k in PiPar is not affected. The FedAvg algorithm is used in PiPar to generate the global model M by aggregating M^k, where $k =$

Algorithm 1: Device-Side Training in PiPar.

```
/* Run on Client $k$.                                    */
Input: local dataset $\mathcal{D}^k$; batch size $B'$; learning rate $\eta$; model split point
       $P^k$; number of mini-batches in each iteration $N^k$
Output: Device-side models $M^{c_k}$
1  Build $M^{c_k}$ based on $P^k$
2  while model has not converged do
       // Start a training epoch
3      for $i = 1$ to $\lfloor|\mathcal{D}^k|/B'N^k\rfloor$ do
           // Start a training iteration
4          for $n = 1$ to $N^k$ do
5              Load a mini-batch $\mathbf{x}_n$ of the size $B'$ from $\mathcal{D}^k$
6              Compute the activation $\mathbf{a}_n$ using Equation (17)
7              Send $\mathbf{a}_n$ and labels $\mathbf{y}_n$ to the server
8          end
9          for $n = 1$ to $N^k$ do
10             Receive $g(\mathbf{a}_n)$ from the server
11             Compute the gradients of model weights $g(M^{c_k}|g(\mathbf{a}_n))$ using
                Equation (22)
12         end
13         Update $M^{c_k} \leftarrow M^{c_k} - \frac{\eta}{N^k}\sum_{n=1}^{N^k} g(M^{c_k}|g(\mathbf{a}_n))$
14     end
15     Send 'stop epoch' signal to the server
16     Send $M^{c_k}$ to the server
17     Receive $M^{c_k}{}'$ from the server
18     Update $M^{c_k} \leftarrow M^{c_k}{}'$
19 end
20 Send 'stop training' signal to the server
21 Return $M^{c_k}$
```

$1, 2, ..., K$, which is the same as in classic FL. Therefore, PiPar maintains a similar model accuracy to FL.

3.2.4. Training overview

The entire training process of PiPar is shown in Algorithm 1 and Algorithm 2.

All devices train simultaneously using Algorithm 1. On device k, the device-side model M^{c_k} is initially built given the split point (Line 1). Line 2 to Line 19 shows the complete training process until the model converges. In each training epoch (Line 3 to Line 18), the entire dataset is processed. A training epoch consists of multiple training iterations, each processing $B'N^k$ data samples. In each training iteration (Line 4 to Line 13), the forward passes of N^k mini-batches are executed in sequence (Line 6), and the activations are sent to the server (Line 7). Their gradients are then received from the server (Line 10), and the backward passes are executed sequentially to compute the gradients of the weights of M^{c_k} (Line 11). At the end of a training iteration, the model is updated based on the gradients (Line 13). After all training iterations are completed, the signal 'stop epoch', and M^{c_k} is sent to the server (Line 15 to Line 16). The device then receives a global device-side model $M^{c_k}{}'$ from the server (Line 17) and uses it to update the current model (Line 18). When the model converges, the client sends a 'stop training' signal to the server, thus completing the training process (Line 20). Since all device-side models are synchronized, all devices will send a 'stop training' signal to the server simultaneously.

Algorithm 2 is executed on the server-side. The server first builds K models M^{s_k}, where $k = 1, 2, ..., K$ (Line 1), and starts training the models until a signal 'stop training' is received from all devices (Line 2). In each training epoch (Line 3 to Line 23), the K models are trained simultaneously (Line 3 to Line 19) and aggregated into a global model (Line 20 to Line 23). A training epoch of model k does not end until a signal 'stop epoch' (Line 5) is received from device k, which involves multiple training iterations. During a training iteration (Line 6 to Line 15), the server receives the activations and labels from device k (Line 7) and uses them to compute the loss function (Line 9 to Line 10). After that, the gradients of activations and model weights are computed (Line

Algorithm 2: Server-Side Training in PiPar.

```
/* Run on the server.                                      */
Input: Number of devices $K$; structure of the DNN with $Q$ layers;
       learning rate $\eta$; model split point $P^k$ and number of
       mini-batches in each iteration $N^k$, where $k = 1, 2, ..., K$
Output: Server-side models $M^{s_k}$, where $k = 1, 2, ..., K$
1  Build $M^{s_k}$, where $k = 1, 2, ..., K$ based on $P^k$
2  while 'stop training' signal not received do
       // Start a training epoch.
3      for $k = 1$ to $K$ in parallel do
4          Initial the size of dataset on device $k$: $D^k \leftarrow 0$
5          while 'stop epoch' signal not received from all devices do
               // Start a training iteration
6              for $n = 1$ to $N^k$ do
7                  Receive activations $\mathbf{a}_n$ and labels $\mathbf{y}_n$
8                  Update $D^k \leftarrow D^k + |\mathbf{a}_n|$
9                  Compute the output $\hat{\mathbf{y}}_n$ using Equation (18)
10                 Compute loss function $l(\mathbf{y}_n, \hat{\mathbf{y}}_n)$
11                 Compute the gradients of activation
                    $g(\mathbf{a}_n) \leftarrow \frac{\partial l}{\partial \hat{\mathbf{y}}_n}\tilde{b}_Q(\tilde{b}_{Q-1}(...\tilde{b}_{Q+1}(\mathbf{a}_n)))$
12                 Compute the gradients of model weights $g(M^{s_k}|a_n)$
                    using Equation (22)
13                 Send $g(\mathbf{a}_n)$ to device $k$
14             end
15             Update $M^{s_k} \leftarrow M^{s_k} - \frac{\eta}{N^k}\sum_{n=1}^{N^k} g(M^{s_k}|a_n)$
16         end
17         Receive $M^{c_k}$ from device $k$
18         Make up complete model $M^k \leftarrow M^{s_k} \oplus M^{c_k}$
19     end
20     Calculate global model $M \leftarrow \sum_{k=1}^{K}\frac{D^k}{\sum_{k=1}^{K} D^k} M^k$
21     Split $M$ to $M^{s_k}{}', M^{c_k}{}'$, where $k = 1, 2, ..., K$ based on $P^k$
22     Send $M^{c_k}{}'$ to device $k$, where $k = 1, 2, ..., K$
23     Update $M^{s_k} \leftarrow M^{s_k}{}'$
24 end
25 Return $M^{s_k}$, where $k = 1, 2, ..., K$
```

11 to Line 12). The former is then sent to device k (Line 13), and the latter is used to update M^{s_k} at the end of the training iteration (Line 15). After receiving the 'stop epoch' signal, the server receives the device-side model M^{c_k} from device k (Line 17) and makes up a complete model M^k (Line 18). The K models M^k, where $k = 1, 2, ..., K$, are aggregated into a global model M (Line 20). M is then split into a server-side model $M^{s_k}{}'$ and a device-side model $M^{c_k}{}'$ (Line 21). $M^{c_k}{}'$ is sent to device k (Line 22), and $M^{s_k}{}'$ is used to update M^{s_k} (Line 23). A training epoch ends. Training is completed when the 'stop training' signal is received from all devices.

If a given device disconnects from the server, the aggregation carried out on the server will exclude the model of the device. If the device reconnects to the server, then it will download the latest global model and continue training.

Algorithm 1 and Algorithm 2 have the same computational complexity as the FedAvg algorithm [28]. However, PiPar introduces parallelism within training.

3.3. Automated parameter selection

To maximize the utilization of idle resources, two parameters of PiPar that impact the performance of the training pipeline are considered:

a) Split point of a DNN is denoted as P. All layers with indices less than or equal to P are deployed on the device and the remaining layers are deployed on the server. The number of layers determines the amount of computation on a server/device, and the volume of data output from the split layer determines the communication traffic. Therefore, finding the most suitable value for P for each device will balance the time required for computation on the server and the device as well as the communication between them.

b) Parallel batch number denoted as N is the number of mini-batches used for concurrent training in each iteration. The computations of the mini-batches fill up the pipeline, so the number of mini-batches for each training iteration must be determined.

The naive choice of $\{P, N\}$ makes the results of PiPar no worse than FL and SFL. When P is the layer number and $N = 1$, PiPar is the same as FL; when P is the same split point as SFL and $N = 1$, PiPar is the same as SFL. However, carefully selected $\{P, N\}$ values can further optimize the performance of PiPar. The optimal values of $\{P, N\}$ can be obtained by an exhaustive search. The model will be trained with all parameter combinations, and then the optimal parameter combination with the shortest training time can be selected. This is unsuitable to be adopted in PiPar in practical as it is time consuming. In addition, we can also select $\{P, N\}$ values empirically. Empirical selections will make PiPar a better solution than FL and SFL, but cannot make it achieve its optimal performance as the exhaustive search. Therefore, we propose an automated parameter selection approach that identifies an optimal or near-optimal combination of parameters in a shorter time than exhaustively searching. These parameters vary with DNNs, server/device combinations, and network conditions. Therefore, the developed approach relies on estimating the training time for different parameters given the DNN and the network condition.

The approach aims to select the best pair of $\{N^k, P^k\}$ for each device k to minimize the idle resources in the three phases. Firstly, we need to know how much they affect the pipeline. Several training iterations are profiled to identify the size of the output data and the training time for each layer of the DNN. Secondly, the training time for each epoch can be estimated using dynamic programming, given a pair of $\{N^k, P^k\}$. Thirdly, the candidates for $\{N^k, P^k\}$ are shortlisted. Since the training time can be estimated for every candidate, the one with the lowest training time will be selected. The three phases are explained in detail as follows.

3.3.1. Phase 1 - profiling

In this phase, an additional training period is required. The complete model is trained on each device and server separately for a predefined number of iterations. If the entire model cannot fit in the memory of the devices, the devices train as many layers as possible and the server trains the complete model. The following information is empirically collected:

a) Time spent in the forward/backward pass of each layer deployed on each device and server. Assume that $\tilde{f}_q^{c_k}$, $\tilde{b}_q^{c_k}$, $\tilde{f}_q^{s}$ and $\tilde{b}_q^{s}$ denote the forward and backward pass of layer q on device k and server, and $t()$ denotes time. Then, $t(\tilde{f}_q^{c_k})$, $t(\tilde{b}_q^{c_k})$, $t(\tilde{f}_q^{s})$ and $t(\tilde{b}_q^{s})$ are the time taken for the forward and backward pass on the devices and server, which are measured and recorded during training.

b) Output data volume of each layer in the forward and backward pass. $\tilde{v}_q^{f}$ and $\tilde{v}_q^{b}$ denote the output data volume for layer q in the forward and backward passes. The data volumes are measured and recorded during training.

3.3.2. Phase 2 - training time estimation

To estimate the time spent in each training epoch of $\{M^{c_k}, M^{s_k}\}$, given the pairs of $\{N^k, P^k\}$ for device k, the time for each training stage must be estimated.

Assume that $f_n^{c_k}$, $b_n^{c_k}$, $f_n^{s_k}$ and $b_n^{s_k}$ is the time spent in the forward and backward passes of M^{c_k} and M^{s_k} for mini-batch n, where $n = 1, 2, ..., N^k$. The time spent in each stage is the sum of the time spent in all relevant layers. Since the size of each mini-batch in PiPar is reduced to $1/N^k$, the time required for each layer is reduced to $1/N^k$. The time of each training stage is estimated by the following:

$$t(f_n^{c_k}) = \sum_{q=1}^{P^k} \frac{t(\tilde{f}_q^{c_k})}{N^k} \tag{5}$$

$$t(f_n^{s_k}) = \sum_{q=P^k+1}^{Q} \frac{t(\tilde{f}_q^{s_k})}{N^k} \tag{6}$$

$$t(b_n^{c_k}) = \sum_{q=1}^{P^k} \frac{t(\tilde{b}_q^{c_k})}{N^k} \tag{7}$$

$$t(b_n^{s_k}) = \sum_{q=P^k+1}^{Q} \frac{t(\tilde{b}_q^{s_k})}{N^k} \tag{8}$$

Assume that u_n^k and d_n^k are the time required for uploading and downloading between device k and the server for mini-batch n, where $n = 1, 2, ..., N^k$, and w_u^k and w_d^k are the uplink and downlink bandwidths. Since the size of transmitted data is reduced to $1/N^k$:

$$t(u_n^k) = \frac{\tilde{v}_{P^k}^{f}}{w_u^k N^k} \tag{9}$$

$$t(d_n^k) = \frac{\tilde{v}_{P^k}^{b}}{w_d^k N^k} \tag{10}$$

The time required by all training stages is estimated using the above equations. The training time of each epoch can be estimated using dynamic programming. Within each training iteration, a given training stage has previous and next stages (exclusions for the first and last stages) as shown in Table 1. The first stage is $f_1^{c_k}$ and the last stage is $b_N^{c_k}$. We use $T(r)$ to denote the total time from the beginning of the training iteration to the end of stage r, and $t(r)$ to denote the time spent in stage r. Thus, the overall training time is $T(b_N^{c_k})$. Since any stage can start only if all of its previous stages have been completed, we have:

$$T(b_N^{c_k}) = t(b_N^{c_k}) + \max_{r \in prev(b_N^{c_k})} T(r) \tag{11}$$

$$T(r) = t(r) + \max_{r' \in prev(r)} T(r') \tag{12}$$

$$T(f_1^{c_k}) = t(f_1^{c_k}) \tag{13}$$

where $prev()$ is the function to obtain all previous stages of the input stage. Since $t(b_N^{c_k})$ is already obtained in Phase 2, Equation (11) to Equation (13) can be solved by recursion. The overall time of one training iteration can then be estimated.

3.3.3. Phase 3 - parameter selection

In this phase, the candidates of $\{N^k, P^k\}$ are shortlisted. Since the training time can be estimated for each candidate, the one with the shortest training time can be selected.

Assume that the DNN has Q layers, such as dense, convolutional and pooling layers, and that the memory of devices can only accommodate the training of Q' layers, where $Q' \le Q$. The range of P^k is $\{P^k | 1 \le P^k \le Q', P^k \in \mathbb{Z}^+\}$, where $\mathbb{Z}^+$ is the set of all positive integers.

Given P^k, the idle time of the device k between the forward pass and backward pass of each mini-batch (the blank timeline between f^c and b^c in Fig. 2(a)) needs to be filled up by the forward passes of the following multiple mini-batches. As a result, the original mini-batch and the following mini-batches are executed concurrently in one training iteration.

For example, as shown in Fig. 2(a), the device idle time between f^c and b^c is equal to $t(u) + t(f^s) + t(b^s) + t(d)$. Thus, the forward passes or backward passes of the subsequent $\lceil \frac{t(u)+t(f^s)+t(b^s)+t(d)}{\min\{t(f^c),t(b^c)\}} \rceil$ mini-batches can be used to fill in the idle time, making the parallel batch number $N = 1 + \lceil \frac{t(u)+t(f^s)+t(b^s)+t(d)}{\min\{t(f^c),t(b^c)\}} \rceil$. Since the batch size used in PiPar is reduced to $1/N$, the time required for forward and backward passes of each layer, uploading and downloading is reduced to $1/N$. The parallel batch number for device k is estimated as:

Table 1
Stages of a training iteration indicating the previous and next stages.

Stage	Previous Stages	Next Stages
$f_1^{c_k}$	n/a	$f_2^{c_k}, u_1^k$
$f_n^{c_k}, 1 < n < N$	$f_{n-1}^{c_k}$	$f_{n+1}^{c_k}, u_n^k$
$f_N^{c_k}$	$f_{N-1}^{c_k}$	$b_1^{c_k}, u_N^k$
u_1^k	$f_1^{c_k}$	$u_2^k, f_1^{s_k}$
$u_n^k, 1 < n < N$	$u_{n-1}^k, f_n^{c_k}$	$u_{n+1}^k, f_n^{s_k}$
u_N^k	$u_{N-1}^k, f_N^{c_k}$	$f_N^{s_k}$
$f_1^{s_k}$	u_1^k	$b_1^{s_k}$
$b_1^{s_k}$	$f_1^{s_k}$	$f_2^{s_k}, d_1^k$
$f_n^{s_k}, 1 < n < N$	$u_n^k, b_{n-1}^{s_k}$	$b_{n+1}^{s_k}$
$b_n^{s_k}, 1 < n < N$	$f_n^{s_k}$	$f_{n+1}^{s_k}, d_n^k$
$f_N^{s_k}$	$u_N^k, b_{N-1}^{s_k}$	$b_N^{s_k}$
$b_N^{s_k}$	$f_N^{s_k}$	d_N^k
d_1^k	$b_1^{s_k}$	$d_2^k, b_1^{c_k}$
$d_n^k, 1 < n < N$	$d_{n-1}^k, b_n^{s_k}$	$d_{n+1}^k, b_n^{c_k}$
d_N^k	$d_{N-1}^k, b_N^{s_k}$	$b_N^{c_k}$
$b_1^{c_k}$	$f_N^{c_k}, d_1^k$	$b_2^{c_k}$
$b_n^{c_k}, 1 < n < N$	$b_{n-1}^{c_k}, d_n^k$	$b_{n+1}^{c_k}$
$b_N^{c_k}$	$b_{N-1}^{c_k}, d_N^k$	n/a

$$
\begin{aligned}
N^k &= 1 + \left\lceil \frac{t(u_n^k) + t(f_n^{s_n}) + t(b_n^{s_k}) + t(d_n^k)}{\min\{t(f_n^{c_k}), t(b_n^{c_k})\}} \right\rceil \\
&= 1 + \left\lceil \frac{\frac{\tilde{v}^f_{P^k}}{w_u^k N^k} + \sum_{q=1}^{P^k} \frac{t(\tilde{f}_q^s)}{N^k} + \sum_{q=P^k+1}^{Q} \frac{t(\tilde{b}_q^s)}{N^k} + \frac{\tilde{v}^b_{P^k}}{w_d^k N^k}}{\min\left\{ \sum_{q=1}^{P^k} \frac{t(\tilde{f}_q^{c_k})}{N^k}, \sum_{q=1}^{P^k} \frac{t(\tilde{b}_q^{c_k})}{N^k} \right\}} \right\rceil \\
&= 1 + \left\lceil \frac{\frac{\tilde{v}^f_{P^k}}{w_u^k} + \sum_{q=1}^{P^k} t(\tilde{f}_q^s) + \sum_{q=P^k+1}^{Q} t(\tilde{b}_q^s) + \frac{\tilde{v}^b_{P^k}}{w_d^k}}{\min\left\{ \sum_{q=1}^{P^k} t(\tilde{f}_q^{c_k}), \sum_{q=1}^{P^k} t(\tilde{b}_q^{c_k}) \right\}} \right\rceil
\end{aligned}
\tag{14}
$$

For each device k, the best $\{N^k, P^k\}$ can be selected from the short-listed candidates by estimating the training time.

Since the training time of PiPar with parameter pair $\{N^k, P^k\}$ is estimated based on profiling data from training complete models with the original batch size, this approach does not guarantee the selection of optimal parameters. However, our experiments in Section 5.4 show that the parameters selected by this approach are similar to optimal values.

4. Convergence analysis

This section analyses the impact of splitting neural network and reordering training stages on model convergence and final accuracy.

4.1. Splitting DNNs and model accuracy

We will demonstrate that splitting a DNN does not impact model accuracy. Assuming that $\mathbf{x}^0$ is a mini-batch of data and $\mathbf{y}$ is the corresponding label set, $\tilde{f}_q$ denotes the forward pass function of layer q and $\mathbf{x}^q$ denotes the output of layer q, where $q = 1, 2, ..., Q$.

$$\mathbf{x}^q = \tilde{f}_q(\mathbf{x}^{q-1}) \tag{15}$$

The forward pass of the complete model M^k in FL is:

$$\hat{\mathbf{y}} = \tilde{f}_Q(\tilde{f}_{Q-1}(...\tilde{f}_1(\mathbf{x}^0))) \tag{16}$$

where $\hat{\mathbf{y}}$ is the output of the final layer. If the model is split, then the training that occurs on the device and server is also split into two phases.

$$\mathbf{a} = \mathbf{x}^P = \tilde{f}_P(\tilde{f}_{P-1}(...\tilde{f}_1(\mathbf{x}))) \tag{17}$$

$$\hat{\mathbf{y}}' = \tilde{f}_Q(\tilde{f}_{Q-1}(...\tilde{f}_{P+1}(\mathbf{a}))) \tag{18}$$

where $\mathbf{a}$ is the activations that are transferred from device k to the server and $\hat{\mathbf{y}}'$ is the final output.

$$
\begin{aligned}
\hat{\mathbf{y}}' &= \tilde{f}_Q(\tilde{f}_{Q-1}(...\tilde{f}_{P+1}(\mathbf{a}))) \\
&= \tilde{f}_Q(\tilde{f}_{Q-1}(...\tilde{f}_1(\mathbf{x}))) \\
&= \hat{\mathbf{y}}
\end{aligned}
\tag{19}
$$

Thus, the loss function when splitting the model is the same as the original loss function when the model is not split.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = l(\mathbf{y}, \hat{\mathbf{y}}') \tag{20}$$

We use $\tilde{b}_q$ to denote the backward pass function of layer q, which is the derivative of $\tilde{f}_q$.

$$\tilde{b}_q(\mathbf{x}^{q-1}) = \frac{\partial \tilde{f}_q(\mathbf{x}^{q-1})}{\partial \mathbf{x}^{q-1}} = \frac{\partial \mathbf{x}^q}{\partial \mathbf{x}^{q-1}} \tag{21}$$

The weights in layer q of the original model and the split model are denoted as $\mathbf{w}_q$ and $\mathbf{w}'_q$, respectively. Assume g is the gradient function, then:

$$g(\mathbf{w}'_q) = \frac{\partial l(\mathbf{y}, \hat{\mathbf{y}}')}{\partial \hat{\mathbf{y}}'} \tilde{b}_Q(\tilde{b}_{Q-1}(...\tilde{b}_{q+1}(\mathbf{x}_q))) \tag{22}$$

$$g(\mathbf{w}_q) = \frac{\partial l(\mathbf{y}, \hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}} \tilde{b}_Q(\tilde{b}_{Q-1}(...\tilde{b}_{q+1}(\mathbf{x}_q))) \tag{23}$$

Based on Equation (22) and Equation (23):

$$g(\mathbf{w}'_q) = g(\mathbf{w}_q) \tag{24}$$

Since splitting a DNN does not change the gradients, it consequently does not impact model accuracy.

4.2. Reordering training stages and model accuracy

We will demonstrate that the model accuracy of a DNN remains the same before and after reordering the training stages. The dataset on client k is denoted as $\mathcal{D}^k$. $\mathcal{B}^k$ denotes a mini-batch in the original training process and $\mathcal{B}_n^k$, where $n = 1, 2, ..., N$, to denote mini-batches in a training round after reordering training stages, where $\mathcal{B}^k = \bigcup_{n=1}^{N} \mathcal{B}_n^k$.

In the original training process, the model is updated after the backward pass of each mini-batch $\mathcal{B}^k$. Assuming M^k is the original model and η is the learning rate, then the updated model is:

$$M_{new}^k = M^k - \frac{\eta}{B} \sum_{\mathbf{x} \in \mathcal{B}^k} g(M^k | \mathbf{x}) \tag{25}$$

In PiPar, the model is updated after the backward pass of the last mini-batch $\mathcal{B}_n^k$ in each training round. The updated model is:

$$M_{new}^k{}' = M^k - \frac{\eta}{N} \sum_{n=1}^{N} g(M^k | \mathcal{B}_n^k) \tag{26}$$

We have:

$$
\begin{aligned}
M_{new}^k{}' &= M^k - \frac{\eta}{N} \sum_{n=1}^{N} \left(\frac{1}{B'} \sum_{\mathbf{x} \in \mathcal{B}_n^k} g(M^k | \mathbf{x}) \right) \\
&= M^k - \frac{\eta}{N B'} \sum_{n=1}^{N} \sum_{\mathbf{x} \in \mathcal{B}_n^k} g(M^k | \mathbf{x}) \\
&= M^k - \frac{\eta}{N B'} \sum_{\mathbf{x} \in \mathcal{B}^k} g(M^k | \mathbf{x}) \\
&\approx M^k - \frac{\eta}{B} \sum_{\mathbf{x} \in \mathcal{B}^k} g(M^k | \mathbf{x}) \\
&= M_{new}^k
\end{aligned}
\tag{27}
$$

Therefore, the updated models with and without reordering the training stages are nearly the same (and the same if $NB' = B$). Thus, reordering training stages does not impact model accuracy.

Table 2
Architecture of VGG-5, ResNet-18 and MobileNetV3-Small [1].

VGG-5	ResNet-18	MobileNetV3-Small
CONV-3-32	CONV-7-64	CONV-3-16
CONV-3-64	RES-3-64 × 2	BNECK-3-16
CONV-3-64	RES-3-128 × 2	BNECK-3-24 × 2
FC-128	RES-3-256 × 2	BNECK-5-40 × 3
FC-X	RES-3-512 × 2	BNECK-5-48 × 2
	FC-X	BNECK-5-96 × 3
		CONV-1-576
		CONV-1-1024
		FC-X

Table 3
Architecture of VGG-16, ResNet-101 and MobileNetV3-Large.

VGG-16	ResNet-101	MobileNetV3-Large
CONV-3-64 × 2	CONV-7-64	CONV-3-16
CONV-3-128 × 2	RES-3-64	BNECK-3-16
CONV-3-256 × 3	RES-3-256 × 3	BNECK-3-24 × 2
CONV-3-512 × 6	RES-3-512 × 4	BNECK-5-40 × 3
FC-4096 × 2	RES-3-1024 × 23	BNECK-3-80 × 4
FC-X	RES-3-2048 × 2	BNECK-3-112 × 2
	FC-X	BNECK-5-160 × 3
		CONV-1-960
		CONV-1-1280
		FC-X

5. Experimental studies

This section quantifies the benefits of PiPar and demonstrates its superiority over existing CML techniques. We first consider the experimental environment in Section 5.1. The training efficiency and the model accuracy and convergence of PiPar are compared against existing CML techniques in Section 5.2 and Section 5.3, respectively. In Section 5.4, the performance of the proposed automated parameter selection approach is evaluated. Section 5.5 analyses the impact of batch size on the performance of PiPar. Section 5.6 explores the impact on performance when using heterogeneous devices, when using differential privacy methods and when the bandwidth changes.

5.1. Setup

The test platform consists of one server and 100 devices. An 8-core i7-11850H processor with 32 GB RAM is used as the server that collaboratively trains DNNs with 100 Raspberry Pi 3B devices, each with 1 GB RAM.

The network conditions considered are: (1) *4G:* 10Mbps uplink bandwidth and 25Mbps downlink bandwidth; (2) *4G+:* 20Mbps uplink bandwidth and 40Mbps downlink bandwidth; (3) *WiFi:* 50Mbps uplink bandwidth and 50Mbps downlink bandwidth. A regular network with a normal error rate is used in the experiments. The TCP/IP protocol used will handle packet loss. When the protocol detects packet loss, it will re-transmit the packet.

Two settings, the first using small DNNs and the second using large DNNs, are used in the experiments. The small DNNs, namely VGG-5 [35], ResNet-18 [13] and MobileNetV3-Small [14] (Table 2) are trained on the MNIST [4] and CIFAR-10 [23,22] datasets. The large DNNs, namely VGG-16 [35], ResNet-101 [13] and MobileNetV3-Large [14] (Table 3) are trained on the CIFAR-100 [22] and Tiny ImageNet [34] datasets. VGG, ResNet and MobileNet series models are convolutional neural networks (CNN) and are representative of high-performing models from the computer vision community for testing CML methods on devices [9,11,37]. Since the Raspberry Pis have limited memory, the large DNNs cannot be trained using FL as the entire model needs to fit on the device memory. The small and large DNNs can be trained using SFL and PiPar since the models are split across the device and server, and the device only executes a few layers. We have chosen a range of small and large DNNs to demonstrate that PiPar can work across a range of settings. MNIST and CIFAR-10 have ten classes, while CIFAR-100 and Tiny ImageNet have 100. Each dataset is split into training, validation and test datasets, as shown in Table 4. During training, the data samples are provided to the DNN as mini-batches. The size of each mini-batch (referred to as batch size), unless otherwise specified, is 100 for each device in FL and SFL. The batch size in PiPar is $\lfloor 100/N^k \rfloor$, where N^k is the parallel batch number for device k and $k = 1, 2, ..., 100$ (refer to Equation (2)).

[1] ‘CONV-A-B’ represents a convolutional layer of A × A kernel size and of B output channels. ‘FC-A’ represents a fully connected layer with the output size A.

Table 4
Training, validation and test data sizes used in the experiments.

Dataset	Training Set Size	Validation Set Size	Test Set Size
MNIST	60,000	2,000	8,000
CIFAR-10	50,000	2,000	8,000
CIFAR-100	50,000	2,000	8,000
Tiny ImageNet	100,000	2,000	8,000

5.2. Efficiency results

The experiments in this section compare the efficiency of PiPar with FL and SFL. Although SL is a popular CML technique, it is significantly slower than SFL since each device operates sequentially. Hence, SL is not considered in these experiments. All possible split points for SFL are benchmarked (based on the benchmarking method adopted in Scission [27]), and the efficiency of SFL with the best split point is reported. The split point and parallel batch number for PiPar are selected by the approach proposed in Section 3.3.

5.2.1. Comparing efficiency

The efficiency of the CML techniques is measured by *training time per epoch*. Section 5.3.1 will highlight that the loss curves of FL, SFL and PiPar overlap, so the same number of epochs are required for model convergence using the three techniques. Hence, if PiPar reduces the training time per epoch, it reduces the overall training time.

Fig. 4 shows the training time per epoch of six DNNs (three small and three large DNNs) for FL, SFL and PiPar under 4G, 4G+ and WiFi network conditions. The three large DNNs, namely VGG-16, ResNet-101 and MobileNetV3-Large cannot be trained using FL as the entire model needs to be trained on the device but the device memory is limited and does not support the size of these models. It is immediately evident that the training time per epoch for PiPar is lower than FL and SFL in all cases.

When training VGG-5 models on MNIST (Fig. 4(a)) and CIFAR-10 (Fig. 4(d)), the difference of FL training time under three network conditions (5.6%) is smaller than that of SFL (59%) and PiPar (57%), because the devices only upload and download once at the end of the training epoch (it requires less communication compared to SFL and PiPar). However, FL trains the entire model on each device, which requires longer computational time. When the bandwidth is low (for example, 4G), FL outperforms SFL, because the latter requires more communication time. However, under 4G+ and WiFi, SFL has shorter training times

‘RES-A-B’ denotes a residual block that consists of two convolutional layers of A × A kernel size and of B output channels. The output of each residual block is the output of the last inner convolutional layer plus the input of the residual block. “BNECK-A-B” denotes a bottleneck residual block that consists of an expansion layer, a convolutional layer with the kernel size A × A and a projection layer with the output channel number B. The number of classes is denoted as X. The activation, batch normalization and pooling layers are not shown for simplicity.

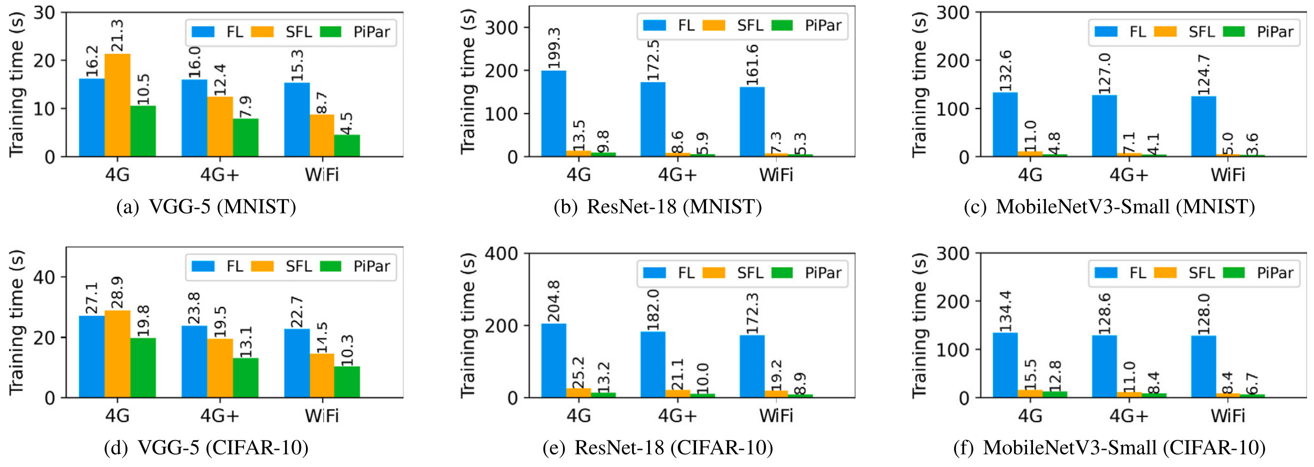


Fig. 4. Training time per epoch for FL, SFL and PiPar under different network conditions for small DNNs.

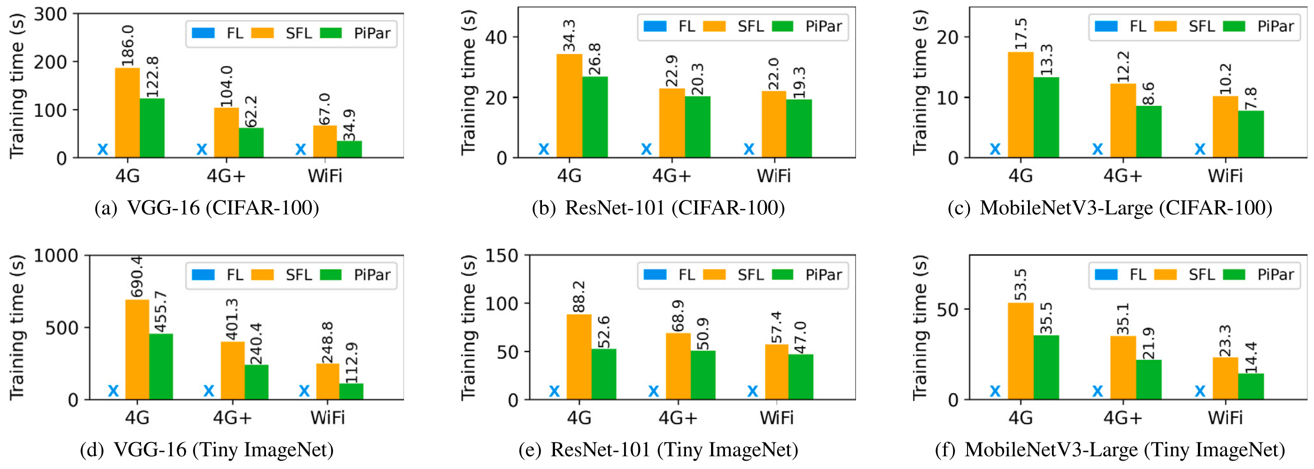


Fig. 5. Training time per epoch for SFL and PiPar under different network conditions for large DNNs. FL results are not shown as the entire DNN does not fit on the device memory.

because of fewer device-side computations. Under all network conditions, PiPar outperforms FL and SFL. It is noteworthy that the benefits of PiPar are evident when training needs to occur in a limited bandwidth environment since more computations can be overlapped with communication (communication takes more time under limited bandwidth). PiPar accelerates FL by 1.4 × - 3.4 × and SFL by 1.4 × - 2.0 ×.

FL is slow when training ResNet-18 (Fig. 4(b) and Fig. 4(e)) and MobileNet-Small (Fig. 4(c) and Fig. 4(f)) because they are deeper networks with more layers. Both SFL and PiPar outperform FL. PiPar has the shortest training time per epoch under all network conditions. In training ResNet-18, PiPar accelerates FL by 15.5 × - 30.5 × and SFL by 1.4 × - 2.2 ×; in training MobileNetV3-Small, PiPar accelerates FL by 10.5 × - 34.6 × and SFL by 1.2 × - 2.3 ×.

Although FL cannot be executed on the devices when training large DNNs, namely VGG-16 (Fig. 5(a) and Fig. 5(d)), ResNet-101 (Fig. 5(b) and Fig. 5(e)) and MobileNetV3-Large (Fig. 5(c) and Fig. 5(f)), PiPar accelerates the training process by 1.12 × - 2.2 × when compared to SFL.

5.2.2. *Comparing resources utilization*

The metric used to compare the utilization of hardware resources is the idle time of the server and devices, which is the total time that the server/device does not contribute to training models in an epoch. The device-side idle time is the average idle time for all devices. A lower idle time corresponds to a higher hardware resource utilization. Since the devices are homogeneous, it is assumed that there is a negligible impact of stragglers.

As shown in Fig. 6, PiPar reduces the server-side idle time under all network conditions when training VGG-5, ResNet-18 and MobileNetV3-Small on MNIST and CIFAR-10. Since the server has more computing resources than the devices, model training is faster on the server. Hence, reducing the server-side idle time takes precedence over reducing the device-side idle time. Since FL trains complete models on the devices, the devices are rarely idle. However, the server is idle for a large proportion of the time when the model is trained. Compared to FL, SFL utilizes more resources on the server because the server trains multiple layers. PiPar reduces the server-side idle time by overlapping the server-side computations, device-side computations and communication between the server and the devices. Compared to FL and SFL, the server-side idle time using PiPar is reduced up to 64.1 × and 2.9 ×, respectively. PiPar also reduces the device-side idle time of SFL up to 23.1 × in all cases.

Fig. 7 highlights that, compared to SFL, PiPar also reduces idle time when training VGG-16, ResNet-101 and MobileNetV3-Large on CIFAR-100 and Tiny ImageNet. Server-side idle time and device-side idle time of SFL are reduced up to 2.3 × and 2.5 ×, respectively.

5.3. *Convergence and model accuracy results*

It is theoretically proven in Section 4 that PiPar achieves comparable model accuracy and convergence as FL. It will be empirically demonstrated that PiPar does not adversely impact the convergence and accuracy of models.

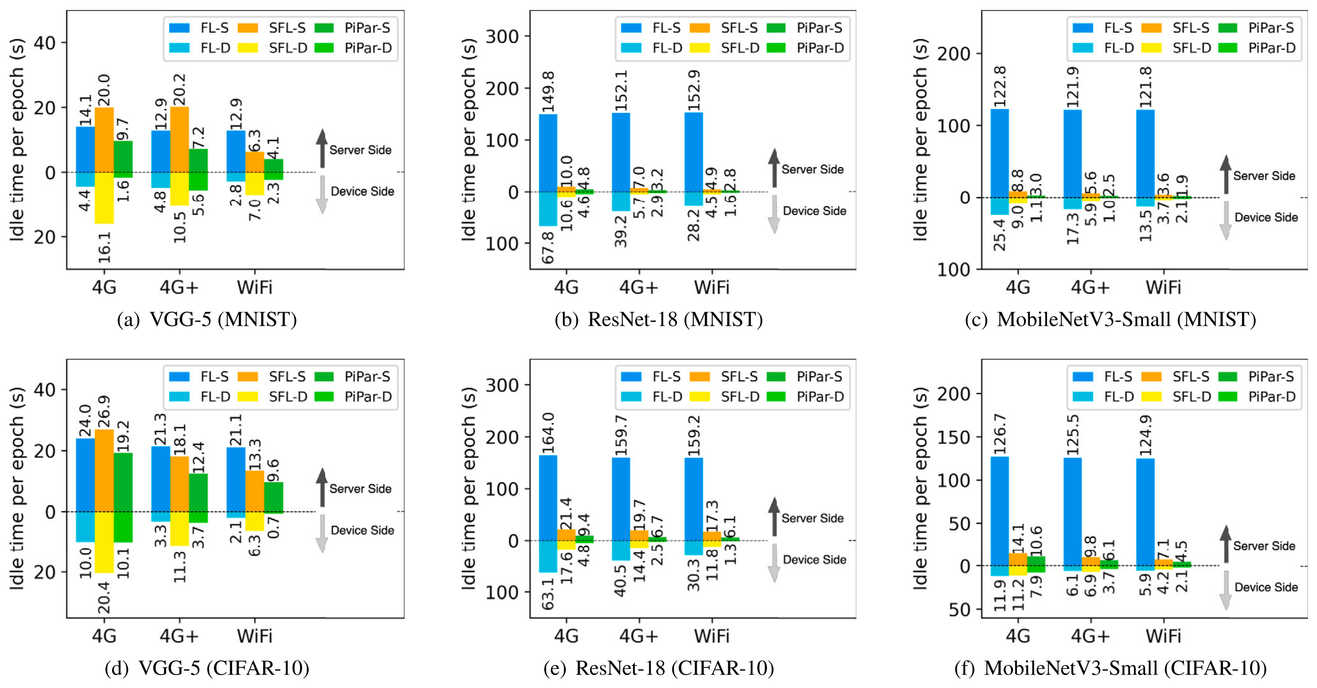


Fig. 6. Idle time per epoch on the server and devices in FL, SFL and PiPar under different network conditions for small DNNs. 'S' and 'D' in the legend represent server-side and device-side idle time, respectively. They are shown in the upward and downward bars.

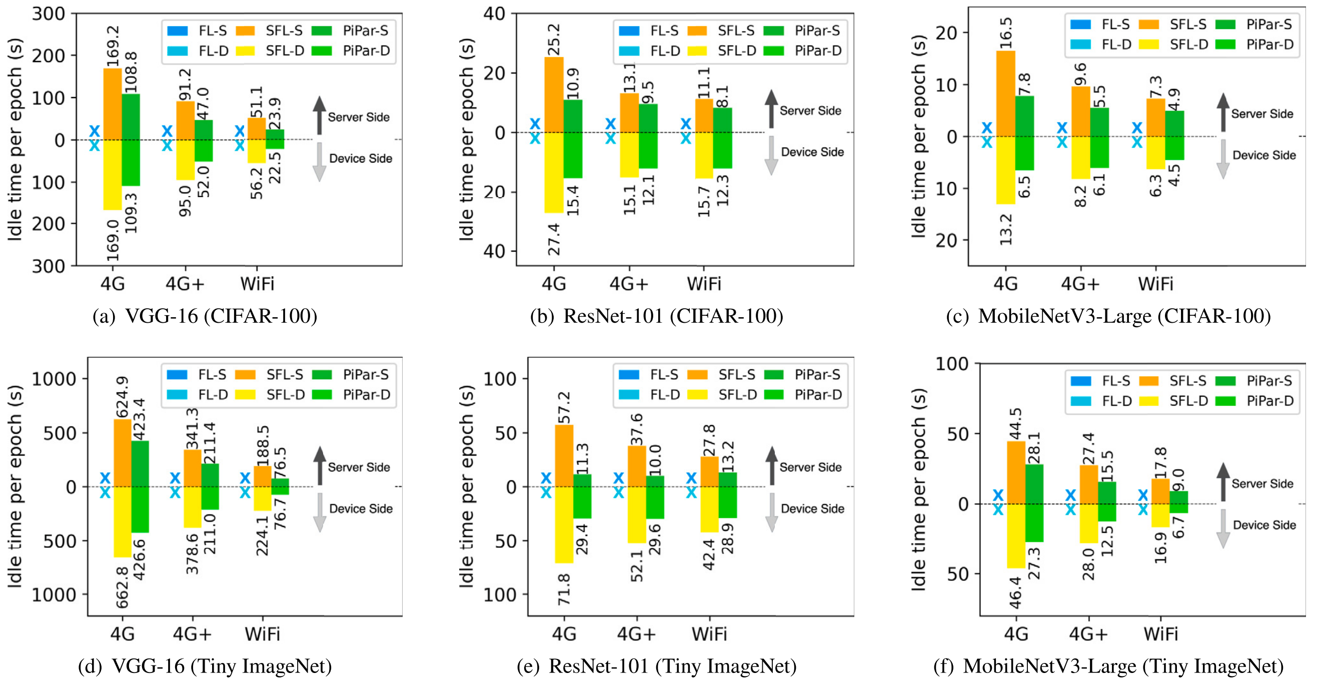


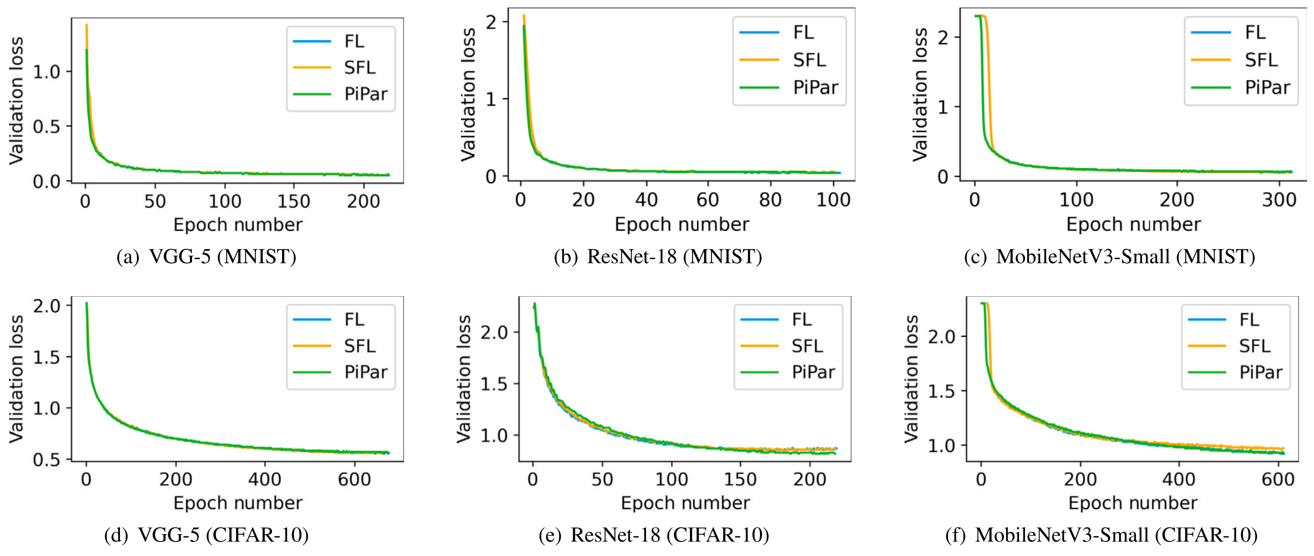
Fig. 7. Idle time per epoch on the server and devices in SFL and PiPar under different network conditions for large DNNs. 'S' and 'D' in the legend represent server-side and device-side idle time, respectively. They are shown in the upward and downward bars. FL results are not shown as the entire DNN does not fit on the device memory.

The convergence curves and test accuracy of the small and large DNNs using FL, SFL and PiPar are reported. Note that due to the limited memory of devices, the large DNNs could not be executed using FL. Since network conditions do not affect model convergence and accuracy in FL and SFL, only the results for WiFi are reported.

5.3.1. Comparing convergence

Fig. 8 and Fig. 9 report the loss curves of FL, SFL and PiPar on the validation datasets using the small and large DNNs, respectively. The results highlight that for all combinations of DNNs and datasets, the loss curves of PiPar generally overlaps those of FL and SFL. Therefore, PiPar does not affect model convergence.

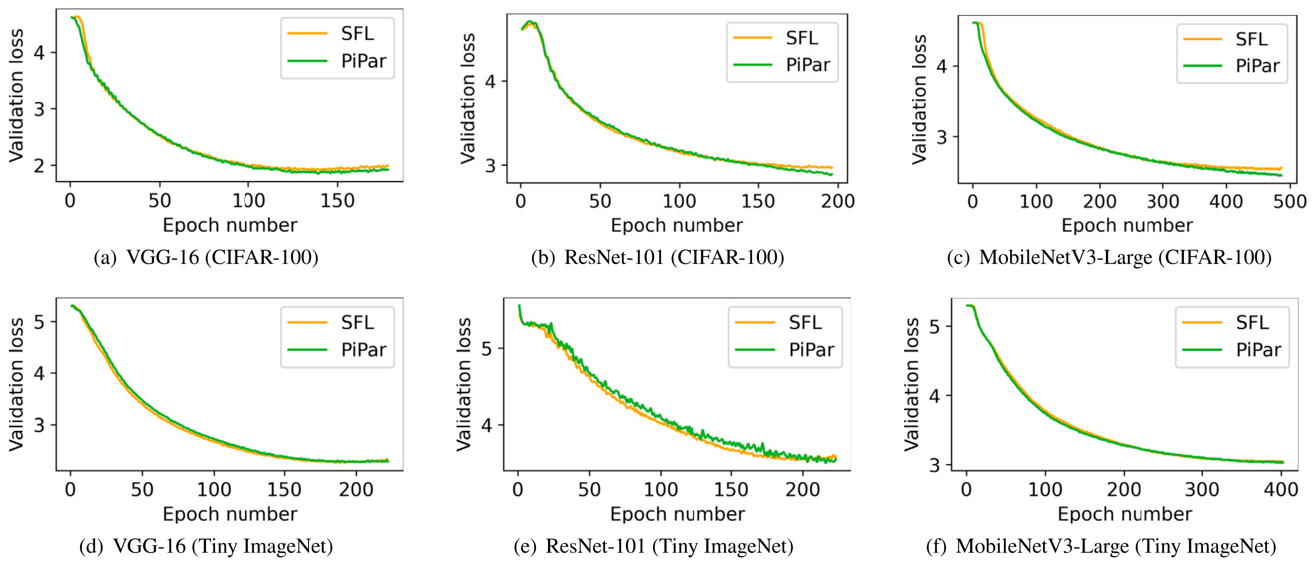
It is noted that regardless of the DNN and dataset choice, PiPar converges within the same number of epochs as FL and SFL. Since PiPar reduces the training time per epoch, as presented in Section 5.2.1, the overall training time is therefore reduced.



(a) VGG-5 (MNIST) (b) ResNet-18 (MNIST) (c) MobileNetV3-Small (MNIST)

(d) VGG-5 (CIFAR-10) (e) ResNet-18 (CIFAR-10) (f) MobileNetV3-Small (CIFAR-10)

Fig. 8. Validation loss for FL, SFL and PiPar using small DNNs.



(a) VGG-16 (CIFAR-100) (b) ResNet-101 (CIFAR-100) (c) MobileNetV3-Large (CIFAR-100)

(d) VGG-16 (Tiny ImageNet) (e) ResNet-101 (Tiny ImageNet) (f) MobileNetV3-Large (Tiny ImageNet)

Fig. 9. Validation loss for SFL and PiPar using large DNNs. FL results are not shown as the entire DNN does not fit on the device memory.

5.3.2. *Comparing accuracy*

In Table 5, the test accuracy of the small DNNs using FL, SFL and PiPar are reported. The last row shows the difference between the model accuracy of PiPar and the higher one of FL and SFL, denoted as Δ. The results for FL,[2] SFL and PiPar on the large DNNs are shown in Table 6. As seen in both tables, in all cases PiPar achieves comparable accuracy as FL and SFL on the test dataset, where the difference in accuracy ranges from -0.2% to +2.07%. Specifically, in the worst case, the test accuracy of training MobileNetV3-Small on CIFAR-10 achieved by PiPar is 0.2 lower than FL but still 0.7 higher than SFL.

These results empirically demonstrate that splitting a DNN and re-ordering the training stages in PiPar does not sacrifice model accuracy while obtaining a higher training efficiency.

[2] Since FL cannot be run on the devices due to limited memory, a high-performance testbed is used to train large DNNs using FL. The high-performance testbed comprising 128 CPUs, two A6000 GPUs and 256 GB memory. Note that the compute capability of devices does not affect the accuracy but only training time.

5.4. *Evaluation of automated parameter selection*

The results presented here demonstrate the effectiveness of the automated parameter selection approach in PiPar. Initially, we exhaustively benchmarked all possible parameters to obtain the optimal parameters. We then show that PiPar selects parameters that are obtained in less time than an exhaustive search, but achieves optimal or near-optimal training time.

The control parameters of PiPar, namely split point P^k and parallel batch number N^k for device k, where $k = 1, 2, ..., K$, affects training efficiency (Section 3.3). P^k and N^k for all devices are the same in our experiments since we consider homogeneous devices; so we use P and N.

The optimal split point P_{opt} and parallel batch number N_{opt} can be found by exhaustively searching given a finite search space. As shown in Table 2 and Table 3, VGG-5, ResNet-18, MobileNetV3-Small, VGG-16, ResNet-101 and MobileNetV3-Large consist of 5, 10, 15, 16, 35 and 19 sequential layers, respectively. We have $P \in [1, 5]$ for VGG-5, $P \in [1, 10]$ for ResNet-18, $P \in [1, 15]$ for MobileNetV3-Small, $P \in [1, 16]$ for VGG-16, $P \in [1, 35]$ for ResNet-101 and $P \in [1, 16]$ for MobileNetV3-Large. Note that DNNs, such as ResNet-18 and ResNet-101, have parallel

Table 5
Model accuracy (percentage) for FL, SFL and PiPar using small DNNs.

Technique	MNIST			CIFAR-10		
	VGG-5	ResNet-18	MobileNetV3-Small	VGG-5	ResNet-18	MobileNetV3-Small
FL	97.96	98.11	97.71	81.39	71.79	67.35
SFL	97.96	98.23	97.71	81.34	71.1	66.45
PiPar	97.94	98.49	97.73	81.31	72.39	67.15
Δ	-0.02	+0.26	+0.02	-0.08	+0.6	-0.2

Table 6
Model accuracy (percentage) for FL, SFL and PiPar using large DNNs.

Technique	CIFAR-100			Tiny ImageNet		
	VGG-16	ResNet-101	MobileNetV3-Large	VGG-16	ResNet-101	MobileNetV3-Large
FL	52.79	26.16	37.01	44.94	23.49	29.3
SFL	51.89	27.11	37.3	44.93	23.49	29.53
PiPar	52.98	27.64	37.64	45.78	25.56	29.6
Δ	+0.19	+0.53	+0.34	+0.84	+2.07	+0.07

branches that cannot be split. In this case, only connections between sequential layers can be selected as split points. Assuming batch size for FL is B, since the batch size for PiPar $\lfloor B/N \rfloor$ is no less than 1, we have $N \in [1, B]$. To exhaustively search for the optimal pair $\{P_{opt}, N_{opt}\}$, the DNNs are trained for one iteration using all possible $\{P, N\}$ pairs in PiPar, and the pair with the shortest training time is considered optimal.

The proposed method is to select P and N for each experiment. There is only one training iteration in the profiling stage. We compare $\{P, N\}$ selected by our approach against $\{P_{opt}, N_{opt}\}$ determined by the exhaustive search in terms of training time and search time. $T_{P,N}$ denotes the training time for each epoch given P and N. We have $T_{P,N} \geq T_{P_{opt},N_{opt}}$. The score in Equation (28) measures how close $T_{P,N}$ is to $T_{P_{opt},N_{opt}}$, which is between 0 and 1. The higher the score, the better the $\{P, N\}$ values perform in terms of training time.

$$score = \frac{T_{P_{opt},N_{opt}}}{T_{P,N}} \tag{28}$$

The results for small and large DNNs are shown in Table 7 and Table 8, respectively. $S_{P,N}$ is the search time to obtain P and N for the automated parameter selection approach or the exhaustive search; smaller is better. For both small and large DNNs, the proposed method selects near optimal parameters in all cases (with a $Score \geq 0.96$) and optimal parameters in 83.3% cases. In addition, our approach selects optimal split point P in all cases.

The results highlight that the time to exhaustively search is substantially high to be practical in the real-world. The average cost of our approach is 27% of one training epoch, which is 6957 × faster than exhaustively searching. The approach is only executed once before training. Since raining consists of hundreds of epochs or more, the overhead of executing this algorithm is negligible (less than 0.3%). Therefore, our approach provides a practical approach to determine the parameters of PiPar.

5.5. Batch size analysis

Compared to FL, PiPar in Phase 2 increases the number of mini-batches involved in each training iteration and reduces the batch size. Assume B is the FL batch size. The batch size in PiPar is $B' = \lfloor B/N \rfloor$, where N is the parallel batch number (Equation (2)). The DNNs are trained using FL and SFL for different B and PiPar for corresponding B' under different network conditions.

Fig. 10 shows the training time per epoch for FL, SFL and PiPar using VGG-5 and CIFAR-10, while Fig. 11 shows the training time per epoch for SFL and PiPar using VGG-16 and CIFAR-100. The training time of FL/SFL/PiPar decreases as the batch size increases because intra-batch parallelisation can be leveraged for matrix multiplication operations when a larger batch is trained. However, increasing batch sizes is not an effective way to speed up training because it requires more memory and reduces model accuracy [18]. The results highlight that PiPar is consistently faster than FL and SFL for VGG-5 and faster than SFL for VGG-16 under all network conditions, regardless of batch sizes. The same trend is seen when training other four DNNs and two datasets.

5.6. Robustness analysis

We explore the robustness of PiPar to more complex environments. In Section 5.6.1, heterogeneous devices are used to evaluate the performance of PiPar against FL and SFL. The impact of using differential privacy methods is considered in Section 5.6.2. Finally, the impact of changing network bandwidth on the overhead of the automated parameter selection approach is considered in Section 5.6.3. In this section, only representative results are shown that are obtained from an evaluation using VGG-5, ResNet-18 and MobileNetV3-Small on CIFAR-10 under different network conditions. A similar trend is noted for other datasets.

5.6.1. Impact on performance with heterogeneous devices

The impact on the performance using a homogeneous and heterogeneous testbed is considered. The setup of the homogeneous testbed was presented in Section 5.1. In the heterogeneous testbed, the same number of devices are used but the CPU frequency of half of the devices is reduced from 1.2 GHz to 600 MHz to create an environment with different compute capabilities of devices.

Fig. 12 shows the training time per epoch for FL, SFL and PiPar. Compared to the homogeneous testbed, the training time on the heterogeneous increases since there are slower devices. The faster devices have to wait for the stragglers before model aggregation. In all cases, PiPar has lowest training time compared to FL and SFL on both homogeneous and heterogeneous testbeds. Specifically, on the heterogeneous testbed, PiPar accelerates training of FL by up to 32 × and SFL by up to 1.8 ×. In addition, FL has a larger difference in performance between testbeds than SFL and PiParsince the latter trains the last several layers of the DNN (as the DNN is partitioned) on the server, which is not affected by the heterogeneity of devices.

5.6.2. Impact on performance when using differential privacy methods

Differential Privacy (DP) [1,24] is used in CML methods to enhance privacy in CML by adding noise into data transferred between the devices and server. We consider the performance overhead introduced when using DP methods in FL, SFL and PiPar.

Two DP methods are considered. Firstly, classic DP [1] is used to add noise to local models on devices before they are sent to the server to make them irreversible. Secondly, PixelDP [24] adds an additional noise layer before the first layer of device models, which prevents activations from being restored to raw data via reverse engineering.

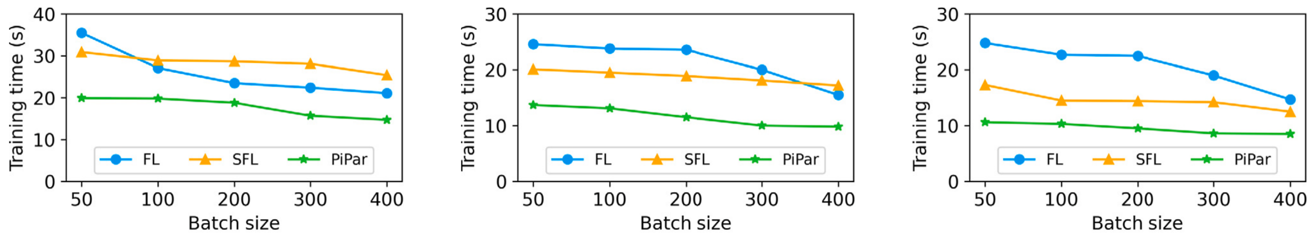
Fig. 13 shows the training time per epoch for different CML methods with and without DP methods. Classic DP introduces an overhead of up

Table 7
Parameters selected by the approach in `PiPar` in contrast to the optimal parameters for small DNNs.

Model	Dataset	Network	Proposed Approach				Exhaustive Search				*Score*
			P	N	$T_{P,N}$	$S_{P,N}$	P_{opt}	N_{opt}	$T_{P_{opt},N_{opt}}$	$S_{P_{opt},N_{opt}}$	(Equation (28))
VGG-5	MNIST	4G	2	4	10.5	1.7	2	4	10.5	481.8	1
		4G+	1	10	7.9	1.4	1	11	7.7	477.0	0.97
		WiFi	1	6	4.5	1.3	1	6	4.5	474.2	1
	CIFAR-10	4G	1	12	19.8	6.2	1	12	19.8	2034.2	1
		4G+	1	6	13.1	4.5	1	8	12.9	1818.4	0.98
		WiFi	1	3	10.3	4.2	1	3	10.3	1816.2	1
ResNet-18	MNIST	4G	1	7	9.8	1.6	1	7	9.8	3393.9	1
		4G+	1	4	5.9	1.5	1	4	5.9	3130.5	1
		WiFi	1	3	5.3	1.4	1	3	5.3	3018.4	1
	CIFAR-10	4G	1	8	13.2	4.2	1	8	13.2	13753.4	1
		4G+	1	6	10.0	4.0	1	6	10.0	13634.0	1
		WiFi	1	3	8.9	3.8	1	3	8.9	13535.9	1
MobileNetV3-Small	MNIST	4G	2	5	4.8	1.2	2	5	4.8	4955.0	1
		4G+	2	4	4.1	1.1	2	4	4.1	4847.5	1
		WiFi	1	4	3.6	1.1	1	4	3.6	4816.7	1
	CIFAR-10	4G	2	8	12.8	4.1	2	8	12.8	33549.2	1
		4G+	1	12	8.4	3.7	1	12	8.4	33382.1	1
		WiFi	1	9	6.7	3.1	1	9	6.7	32780.5	1

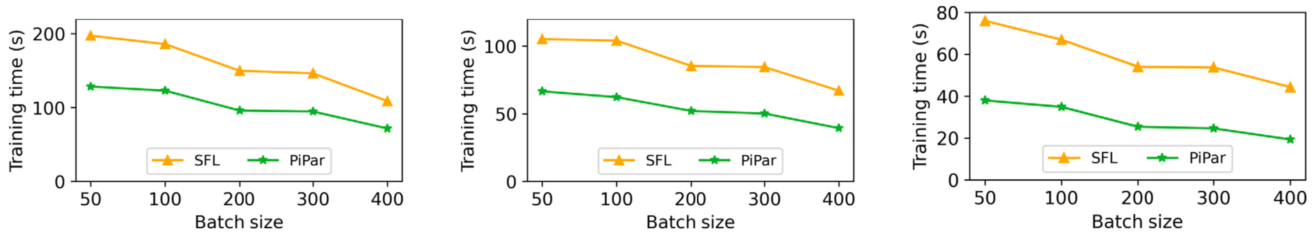
Table 8
Parameters selected by the approach in `PiPar` in contrast to the optimal parameters for large DNNs.

Model	Dataset	Network	Proposed Approach				Exhaustive Search				*Score*
			P	N	$T_{P,N}$	$S_{P,N}$	P_{opt}	N_{opt}	$T_{P_{opt},N_{opt}}$	$S_{P_{opt},N_{opt}}$	(Equation (28))
VGG-16	CIFAR-100	4G	1	20	122.8	13.6	1	16	117.6	34274.4	0.96
		4G+	1	16	62.2	6.1	1	16	62.2	19753.2	1
		WiFi	1	6	34.9	5.2	1	6	34.9	12535.2	1
	Tiny ImageNet	4G	1	16	455.7	22.4	1	16	455.7	824884.0	0.96
		4G+	1	14	240.4	18.7	1	16	233.7	452372.8	0.97
		WiFi	1	12	112.9	16.8	1	16	110.2	245357.0	1
ResNet-101	CIFAR-100	4G	1	4	26.8	5.9	1	4	26.8	29004.6	1
		4G+	1	2	20.3	4.3	1	2	20.3	28408.6	1
		WiFi	1	2	19.3	3.8	1	2	19.3	28347.6	1
	Tiny ImageNet	4G	1	8	52.6	11.1	1	8	52.6	122736.22	1
		4G+	1	6	50.9	10.3	1	4	49.0	119846.0	0.96
		WiFi	1	3	47.0	9.9	1	3	47.0	117718.5	1
MobileNetV3-Large	CIFAR-100	4G	1	16	13.3	4.1	1	16	13.3	41844.5	1
		4G+	1	6	8.6	3.6	1	6	8.6	36807.8	1
		WiFi	1	4	7.8	3.4	1	4	7.8	35651.0	1
	Tiny ImageNet	4G	1	16	35.5	8.6	1	16	35.5	79864.6	1
		4G+	1	12	21.9	7.8	1	12	21.9	57710.0	1
		WiFi	1	8	14.4	7.7	1	8	14.4	46406.2	1



(a) 4G (b) 4G+ (c) WiFi

Fig. 10. Training time per epoch for FL, SFL and `PiPar` using VGG-5 and the CIFAR-10 dataset with different batch sizes B.



(a) 4G (b) 4G+ (c) WiFi

Fig. 11. Training time per epoch for SFL and `PiPar` using VGG-16 and the CIFAR-100 dataset with different batch sizes B. FL results are not shown as the entire DNN does not fit on the device memory.

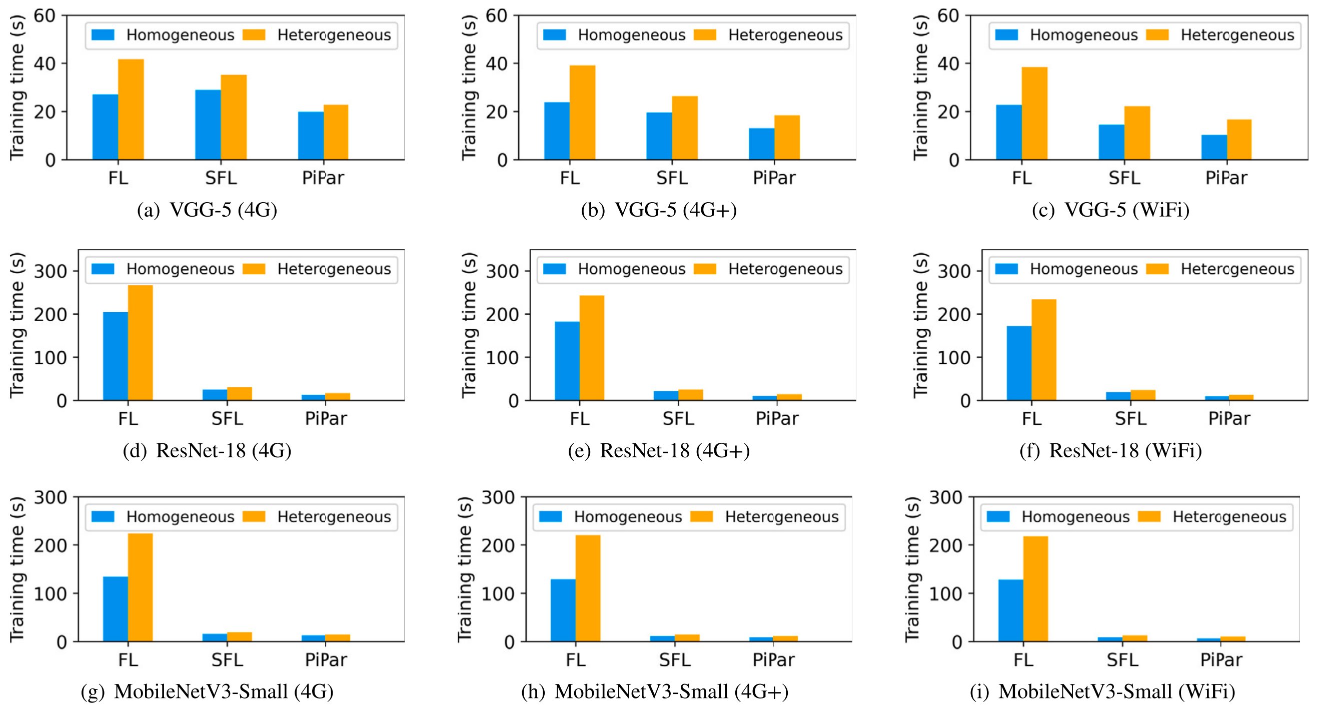


Fig. 12. Training time per epoch for FL, SFL and PiPar using small DNNs on CIFAR-10 under different network conditions on homogeneous and heterogeneous testbeds.

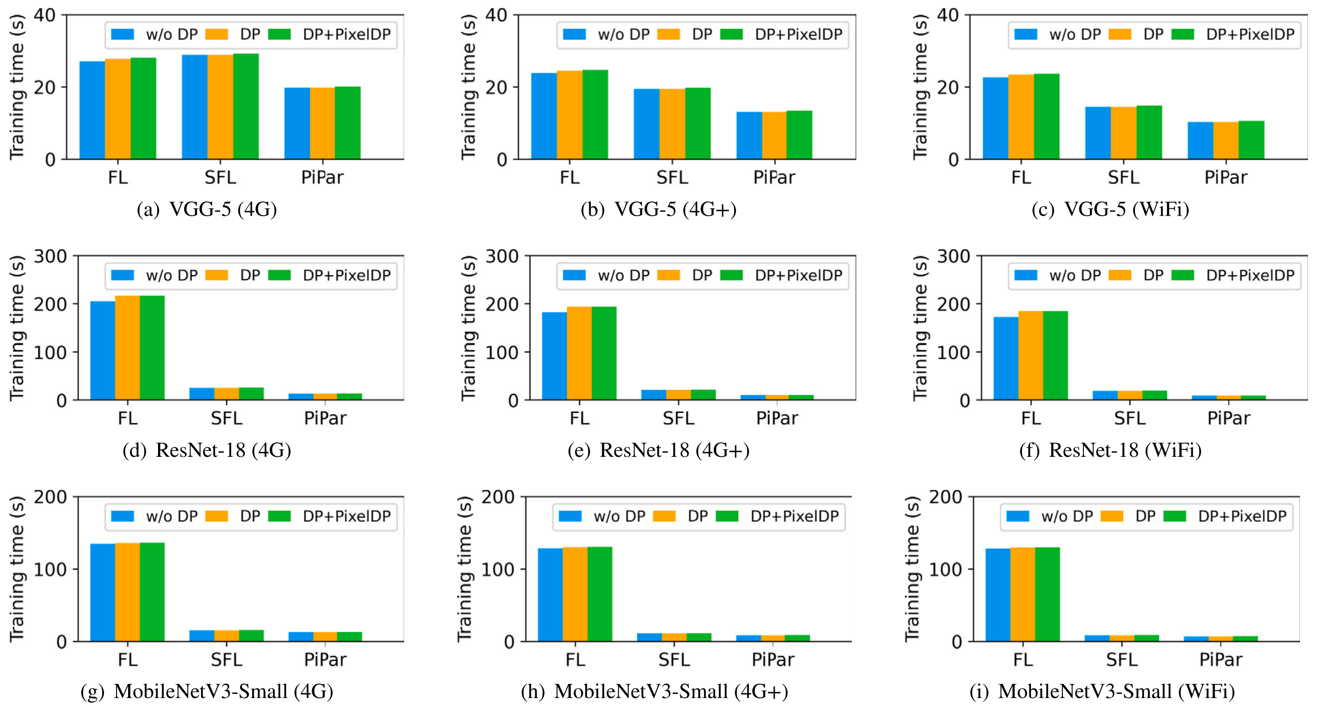


Fig. 13. Training time per epoch for FL, SFL and PiPar using small DNNs on CIFAR-10 under different network conditions with and without differential privacy methods.

to 11.7 s to FL, 0.16 s to SFL and 0.15 s to PiPar. Compared to SFL and PiPar, FL has the largest overhead using DP because in FL the entire model is trained on the device and classic DP will add noise to each parameter in the model. The overhead of PixelDP on FL, SFL and PiPar (up to 0.3 s) is comparable since they use the same size of inputs and PixelDP adds noise to these inputs. The results highlight that the two DP methods applied to PiPar do not introduce a larger overhead than FL and SFL.

5.6.3. *Impact of changing bandwidth on the overhead for automated parameter selection*

It was shown in Section 5.4 that the automated parameter selection approach only needs to execute once before training in a stable network environment. Therefore, the overhead incurred is negligible. In this section, we measure the overhead of the approach in an unstable network where the bandwidth changes between 4G, 4G+ and WiFi conditions periodically in a controlled manner.

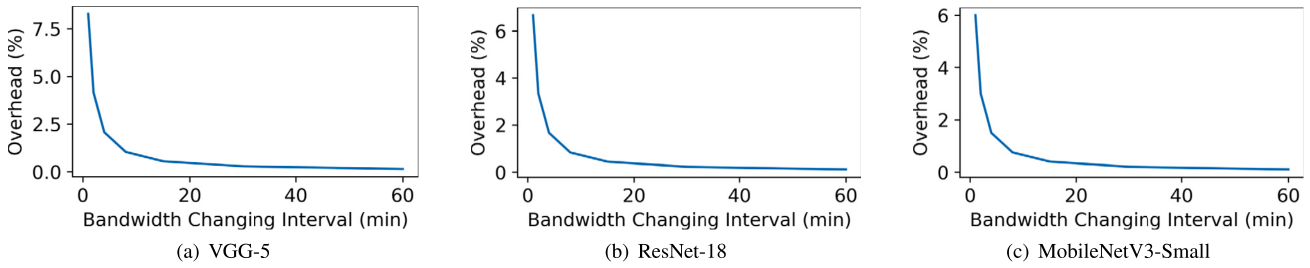


Fig. 14. The percentage overhead of the automated parameter selection approach with respect to training time for different intervals in which the bandwidth changes.

The overhead is measured for different intervals in which the bandwidth changes. The network is more unstable for smaller intervals. Fig. 14 shows the percentage overhead for running the parameter selection approach with respect to training time for different intervals in which the bandwidth changes. The intervals considered range from 1 minute to 60 minutes. If the bandwidth changes every hour, the overhead of parameter selection is only 0.14%, 0.11% and 0.1% of the training time for VGG-5, ResNet-18 and MobileNetV3-Small, respectively. Considering the worst case in the experiments, which is a change of bandwidth every minute, the approach overhead is up to 8.3% of the training time. If the bandwidth change occurs on an average every 10 minutes or more then the overhead incurred is less than 1%.

6. Conclusion

Deep learning models are collaboratively trained using paradigms, such as federated learning, split learning or split federated learning on a server and multiple devices. However, they are limited in that the computation and communication across the server and devices are inherently sequential. This results in low compute and network resource utilization and leads to idle time on the resources. We propose a novel framework, PiPar, that addresses this problem for the first time by taking advantage of pipeline parallelism, thereby accelerating the entire training process. A novel training pipeline is developed to parallelize server-side and device-side computations as well as server-device communication. In the training pipeline, the DNN is split and deployed on the server and devices, and the training process on different mini-batches of data is re-ordered. A low overhead parameter selection approach is then proposed to maximize the resource utilization of the pipeline. Consequently, when compared to existing paradigms, our pipeline significantly reduces idle time on compute resources by up to 64.1 × in training popular DNNs under different network conditions. An overall training speed up of up to 34.6 × is observed. It is also experimentally demonstrated that PiPar achieves performance benefits when incorporating differential privacy methods and operating in environments with heterogeneous devices and changing bandwidths.

CRediT authorship contribution statement

Zihan Zhang: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Philip Rodgers:** Writing – review & editing, Conceptualization. **Peter Kilpatrick:** Writing – review & editing, Conceptualization. **Ivor Spence:** Writing – review & editing, Conceptualization. **Blesson Varghese:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Zihan Zhang reports financial support was provided by Rakuten Mobile Inc., Japan. Blesson Varghese reports financial support was provided by Rakuten Mobile Inc., Japan. Zihan Zhang, Blesson Varghese, Philip Rodgers, Ivor Spence, Peter Kilpatrick has patent pending to Rakuten Mobile Inc., Japan. Co-author Philip Rodgers was employed by Rakuten Mobile Inc., Japan. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

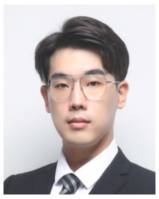
Acknowledgments

This work was sponsored by Rakuten Mobile, Inc., Japan.

References

[1] M. Abadi, A. Chu, I. Goodfellow, H.B. McMahan, I. Mironov, K. Talwar, L. Zhang, Deep learning with differential privacy, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 308–318.
[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: Advances in Neural Information Processing Systems, 2020, pp. 1877–1901.
[3] X. Chen, J. Li, C. Chakrabarti, Communication and computation reduction for split learning using asynchronous training, in: IEEE Workshop on Signal Processing Systems, 2021, pp. 76–81.
[4] L. Deng, The MNIST database of handwritten digit images for machine learning research, IEEE Signal Processing Magazine 29 (2012) 141–142.
[5] J. Devlin, M.W. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, in: Conference of the North American Chapter of the Association forComputational Linguistics: Human Language Technologies, 2019, pp. 4171–4186.
[6] Y. Gao, M. Kim, S. Abuadbba, Y. Kim, C. Thapa, K. Kim, S.A. Camtep, H. Kim, S. Nepal, End-to-end evaluation of federated learning and split learning for Internet of things, in: International Symposium on Reliable Distributed Systems, 2020, pp. 91–100.
[7] A. Graves, Sequence transduction with recurrent neural networks, in: International Conference of Machine Learning Workshop, 2012.
[8] O. Gupta, R. Raskar, Distributed learning of deep neural network over multiple agents, J. Netw. Comput. Appl. 116 (2018) 1–8.
[9] D.J. Han, D.Y. Kim, M. Choi, C.G. Brinton, J. Moon, Splitgp: achieving both generalization and personalization in federated learning, in: IEEE International Conference on Computer Communications, 2023, pp. 1–10.
[10] A.Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, A.Y. Ng, Deep speech: scaling up end-to-end speech recognition, CoRR, arXiv:1412.5567 [abs], 2014.
[11] C. He, M. Annavaram, S. Avestimehr, Group knowledge transfer: federated learning of large CNNs at the edge, in: Proceedings of the 34th International Conference on Neural Information Processing Systems, 2020.
[12] K. He, G. Gkioxari, P. Dollár, R. Girshick, Mask R-CNN, in: IEEE International Conference on Computer Vision, 2017, pp. 2980–2988.
[13] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[14] A. Howard, M. Sandler, G. Chu, L. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q.V. Le, H. Adam, Searching for MobileNetV3, in: IEEE/CVF International Conference on Computer Vision, 2019, pp. 1314–1324.
[15] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q.V. Le, Y. Wu, z. Chen, GPipe: efficient training of giant neural networks using pipeline parallelism, in: Advances in Neural Information Processing Systems, 2019.

[16] Z. Ji, L. Chen, N. Zhao, Y. Chen, G. Wei, F.R. Yu, Computation offloading for edge-assisted federated learning, IEEE Trans. Veh. Technol. 70 (2021) 9330–9344.
[17] A. Kendall, Y. Gal, What uncertainties do we need in Bayesian deep learning for computer vision?, in: 31st International Conference on Neural Information Processing Systems, 2017, pp. 5580–5590.
[18] N.S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P.T.P. Tang, On large-batch training for deep learning: generalization gap and sharp minima, in: International Conference on Learning Representations, 2017.
[19] J. Konečný, B. McMahan, D. Ramage, Federated Optimization: Distributed Optimization Beyond the Datacenter, in: 8th NIPS Workshop on Optimization for Machine Learning, 2015.
[20] J. Konečný, H.B. McMahan, D. Ramage, P. Richtárik, Federated Optimization: Distributed Machine Learning for On-Device Intelligence, CoRR, abs/1610.02527, 2016.
[21] J. Konečný, H.B. McMahan, F.X. Yu, P. Richtárik, A.T. Suresh, D. Bacon, Federated learning: strategies for improving communication efficiency, CoRR, arXiv:1610.05492 [abs], 2016.
[22] A. Krizhevsky, G. Hinton, Learning Multiple Layers of Features from Tiny Images, Master's thesis, Department of Computer Science, University of Toronto, 2009.
[23] A. Krizhevsky, V. Nair, G. Hinton, Canadian Institute for Advanced Research, 2009, CIFAR-10.
[24] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, S. Jana, Certified robustness to adversarial examples with differential privacy, in: 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 656–672.
[25] S. Li, T. Hoefler, Chimera: efficiently training large-scale neural networks with bidirectional pipelines, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.
[26] J. Liu, H. Xu, L. Wang, Y. Xu, C. Qian, J. Huang, H. Huang, Adaptive asynchronous federated learning in resource-constrained edge computing, IEEE Trans. Mob. Comput. 22 (2023).
[27] L. Lockhart, P. Harvey, P. Imai, P. Willis, B. Varghese, Scission: context-aware and performance-driven edge-based distributed deep neural networks, in: 13th IEEE/ACM International Conference on Utility and Cloud Computing, 2020.
[28] B. McMahan, E. Moore, D. Ramage, S. Hampson, B.A.y. Arcas, Communication-efficient learning of deep networks from decentralized data, in: 20th International Conference on Artificial Intelligence and Statistics, 2017, pp. 1273–1282.
[29] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N.R. Devanur, G.R. Ganger, P.B. Gibbons, M. Zaharia, PipeDream: generalized pipeline parallelism for DNN training, in: 27th ACM Symposium on Operating Systems Principles, 2019, pp. 1–15.
[30] T. Nishio, R. Yonetani, Client selection for federated learning with heterogeneous resources in mobile edge, in: IEEE International Conference on Communications, 2019, pp. 1–7.
[31] K. Osawa, S. Li, T. Hoefler, PipeFisher: efficient training of large language models using pipelining and Fisher information matrices, in: Proceedings of Machine Learning and Systems, 2023.
[32] S. Pal, M. Uniyal, J. Park, P. Vepakomma, R. Raskar, M. Bennis, M. Jeon, J. Choi, Server-side local gradient averaging and learning rate acceleration for scalable split learning, CoRR, arXiv:2112.05929 [abs], 2021.
[33] Z. Qu, S. Guo, H. Wang, B. Ye, Y. Wang, A. Zomaya, B. Tang, Partial synchronization to accelerate federated learning over relay-assisted edge networks, IEEE Trans. Mob. Comput. (2021) 1.
[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M.S. Bernstein, A.C. Berg, L. Fei-Fei, ImageNet large scale visual recognition challenge, CoRR, arXiv:1409.0575 [abs], 2014.
[35] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: 3rd International Conference on Learning Representations, 2015, pp. 1–14.
[36] A. Singh, P. Vepakomma, O. Gupta, R. Raskar, Detailed comparison of communication efficiency of split learning and federated learning, CoRR, arXiv:1909.09145 [abs], 2019.
[37] C. Thapa, M.A.P. Chamikara, S. Camtepe, SplitFed: when federated learning meets split learning, AAAI Conf. Artif. Intell. 36 (8) (2022) 8485–8493.
[38] C. Thapa, M.A.P. Chamikara, S.A. Camtepe, Advancements of federated learning towards privacy preservation: from federated learning to split learning, in: M.H.u. Rehman, M.M. Gaber (Eds.), Federated Learning Systems: Towards Next-Generation AI, Springer International Publishing, 2021, pp. 79–109.
[39] P. Vepakomma, O. Gupta, T. Swedish, R. Raskar, Split learning for health: distributed deep learning without sharing raw patient data, in: ICLR Workshop on AI for Social Good, 2019.
[40] Z. Wang, H. Xu, J. Liu, Y. Xu, H. Huang, Y. Zhao, Accelerating federated learning with cluster construction and hierarchical aggregation, IEEE Trans. Mob. Comput. (2022) 1.
[41] D. Wu, R. Ullah, P. Harvey, P. Kilpatrick, I. Spence, B. Varghese, FedAdapt: adaptive offloading for IoT devices in federated learning, IEEE Int. Things J. 9 (2022) 20889–20901.
[42] W. Xu, W. Fang, Y. Ding, M. Zou, N. Xiong, Accelerating federated learning for IoT in big data analytics with pruning, quantization and selective updating, IEEE Access 9 (2021) 38457–38466.
[43] Z. Xu, F. Yu, J. Xiong, X. Chen, Helios: heterogeneity-aware federated learning with dynamically balanced collaboration, in: 58th ACM/IEEE Design Automation Conference, 2021, pp. 997–1002.
[44] Z. Xu, Y. Zhang, G. Andrew, C. Choquette, P. Kairouz, B. Mcmahan, J. Rosenstock, Y. Zhang, Federated learning of gboard language models with differential privacy, in: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, 2023, pp. 629–639.
[45] B. Yang, J. Zhang, J. Li, C. Ré, C.R. Aberger, C.D. Sa, PipeMare: asynchronous pipeline parallel DNN training, in: Seventh Conference on Machine Learning and Systems, 2024.
[46] Y. Ye, S. Li, F. Liu, Y. Tang, W. Hu, EdgeFed: optimized federated learning based on edge computing, IEEE Access 8 (2020) 209191–209198.
[47] R. Yu, P. Li, Toward resource-efficient federated learning in mobile edge computing, IEEE Netw. 35 (2021) 148–155.
[48] Y. Zhou, Q. Ye, J. Lv, Communication-efficient federated learning with compensated overlap-FedAvg, IEEE Trans. Parallel Distrib. Syst. 33 (2022) 192–205.


Zihan Zhang is a PhD student at University of St Andrews. He was previously a researcher at Noah's Ark Lab, Huawei. Prior to that, he obtained a Master's degree in Data Science with distinction from University of Edinburgh and a Bachelor's degree in Mathematics from Shandong University. His interests include collaborative machine learning and cloud/edge computing.

Philip Rodgers received the PhD degree in Computer Science from the University of Strathclyde, UK in 2019. He was a Research Scientist at the Rakuten Mobile Innovation Studio, Japan when this work was carried out. His research interests are in distributed algorithms for artificial intelligence.

Peter Kilpatrick received the PhD degree in Computer Science in 1985, and the BSc degree in Mathematics and Computer Science in 1981. He is currently a Reader in Computer Science at Queen's University Belfast. His interests include parallel programming models and cloud and edge computing.

Ivor Spence received the PhD degree in computer science from Queen's University Belfast, UK, where he did research on code generation. He is currently a Reader in computer science at Queen's University Belfast where he leads the artificial intelligence (AI) research theme. His research is primarily on heterogeneous computing systems for AI.

Blesson Varghese received the PhD degree in computer science from the University of Reading, UK. He is a Reader in computer science at the University of St Andrews, UK, and the Principal Investigator of the Edge Computing Hub. He was a Royal Society Short Industry Fellow to British Telecommunications plc. His interests include distributed systems that span the cloud-edge-device continuum and edge intelligence applications.