Towards Accurate Training Time Estimation for On-Device Heterogeneous Federated Learning

Kun Wang, Zimu Zhou, *Member, IEEE,* and Zhenjiang Li, *Member, IEEE*

Abstract—Accurate estimation of on-device model training time is increasingly required for emerging learning paradigms on mobile edge devices, such as heterogeneous federated learning (HFL). HFL usually customizes the model architecture according to the different capabilities of mobile edge devices to ensure efficient use of local data from all devices for training. However, due to oversimplification of training time modeling, existing methods rely on a single coefficient to represent computational heterogeneity, resulting in sub-optimal HFL efficiency. We find that existing methods ignore the important impact of runtime optimization of deep learning frameworks, which we call development-chain diversity. Specifically, layers of a model may have different algorithm implementations, and deep learning frameworks often have different strategies for selecting the algorithm they believe is the best based on a range of runtime factors, resulting in different training latencies and invalid predictions from existing methods. In this paper, in addition to considering this diversity to ensure synchronized completion time of model training, we also study how to select the best algorithm each time to reduce the latency of the per-round training, thereby further improving the overall efficiency of federated training. To this end, we propose LATTE, which consists of a novel data-driven selector that identifies the best algorithm at runtime based on relative runtime factors. By further integrating it into our training latency model, LATTE provides accurate training time estimation, significantly outperforming traditional heuristic approaches. To further improve the robustness of LATTE, we proposed dynamic device adapter to cope with the dynamic joining and exiting of the clients. We develop LATTE as middleware, compatible with different deep learning frameworks. Extensive results show significantly improved training convergence speed and model accuracy compared to state-of-the-art methods.

Index Terms—On-Device Learning, Heterogeneous Federated Learning, Training Time Estimation

1 Introduction

As mobile edge devices such as NVIDIA Jetson series becoming increasingly powerful and IoT applications [45], [30], [18], [78], [77] becoming more advanced, these devices not only can efficiently perform model *inference* tasks [70], [67], [69], [19], [75], but also can perform certain model *training* tasks [26]. However, since the computing capability of mobile edge devices is far less than that of personal computers or servers, on-device training at the mobile edge is not positioned to replace traditional model training. It mainly focuses on updating the model according to the requirements of many emerging learning paradigms, such as federated learning (FL) [24], [38], [40], [54], [55], [63], [66], [62].

Training time is a key factor in determining the efficiency of federated learning on mobile edge devices as different devices may have vastly different computing and communication capabilities, known as *system heterogeneity* [81]. Specifically, multiple edge devices (as clients) collaborate to train a model under the coordination of the server [52]. In each round of training, the server assigns the current model to the devices. Each device then trains the model using its own data and returns updates to the server, which are aggregated into the next round of models. Primarily due to the heterogeneity in *computing capability* [56], devices usually cannot complete training of the same model at the same time, and the server must wait for updates of the slowest device in each round [52], leading to significant overall training latency measured in wall-clock time. This latency can seriously harm the availability of federated learning systems, especially in time-sensitive scenarios such as personal assistants, drone fleets, and robot swarms [23], [51], [73].

When the number of federated devices is large (*e.g.*, hundreds or thousands), recent work has found that training efficiency can be improved through client selection [37], [40], [43], [47] or asynchronous FL [74], [63], mitigating the impact of weak devices on training. However, a more general solution, known as heterogeneous federated learning (HFL), is to allocate appropriate sub-models according to the computing capability of the device [14], [25], [59], [24], [54], [48]. The HFL literature shows that having each device complete each round of model training simultaneously as much as possible can lead to better performance [76], [42]. This not only utilizes the training data on each participating device, but also significantly improves the efficiency of the overall federated training. It is particularly useful in most mobile edge applications in practice with medium-sized device networks, where each device may contain a unique set of data.

Since training time is critical for federated learning, the training time for each device needs to be accurately estimated to allocate appropriate sub-models. To this end, existing work typically predicts the latency by performing linear regression on the computational load of the model [25], [24], such as the number of floating point operations (FLOPs). They characterize computational heterogeneity across devices by computing capability

- *K. Wang and Z. Li are with the Department of Computer Science, City University of Hong Kong, Hong Kong, China. E-mail: kwang69@cityu.edu.hk, zhenjiang.li@cityu.edu.hk.*
- *Z. Zhou is with the Department of Data Science, City University of Hong Kong, Hong Kong, China. E-mail: zimuzhou@cityu.edu.hk.*

related coefficients (*e.g.*, float point per second (FLOPS)) determined offline. Some recent studies have further proposed finer-grained layer-wise modeling [14], [48], acknowledging that different layer types may exhibit different execution times on the same platform but follow the same estimation principle. In this paper, we find that existing approaches ignore an important impact of different deep learning frameworks (*e.g.*, PyTorch and TensorFlow) exhibit distinct behaviors in invoking acceleration libraries (*e.g.*, cuDNN) during runtime optimization [80], [49], [44], which we refer to as *development-chain diversity*. It can cause significant deviations in training latency for the same model, even on the same device. This can make the characterization based on hardware parameters learned in advance inaccurate.

By delving into the runtime behaviors of deep learning frameworks, we discover that the core reason is the diversity of layer algorithms. For example, the convolutional layers, essential in many neural networks, can be implemented through various algorithms, such as direct convolution, matrix multiplication, and fast Fourier transform. However, different deep learning frameworks have different strategies to select what they think is the best layer algorithm at runtime based on hardware specifications, layer configuration, and available runtime resources. This selection needs to be made for both *forward* and *backward* propagation of layers before training, so the choice of layer algorithm can significantly affect the overall training latency. As shown in § 2.3, training the same model can have a 30.1% time difference and be 2.68× slower than the optimal for this reason.

If we deliberately measure the latency differences caused by the diversity of algorithm choices at each layer and take this into account when assigning sub-models, different devices can achieve synchronized completion times of model training. However, in addition to uncovering this key question, we aim to go one step further: Can every device also choose the best algorithm each time? If possible, training can not only be completed synchronously across devices, but the latency in each round can also be minimized (so that we can allocate the largest sub-model), which will further improve the overall training efficiency and performance [25].[1]

To this end, we propose LATTE, a Layer Algorithm-aware Training Time Estimator for HFL, which adapts to the diversity of layer algorithms in the development chain to achieve accurate training time estimation. A key challenge in designing LATTE is that layer algorithm selection is affected by certain runtime parameters, such as resource availability and model configuration, which results in the inability to enumerate the consequences of choices in advance.

To solve this problem, LATTE provides a lightweight and accurate layer algorithm selector. A main novelty in its design is the generation of training data and ground truth for the selector. Specifically, we design a acquisition tool that can derive a set of training data and leverage the functional modules from existing deep learning frameworks to generate corresponding ground truth for training the selector in a device-independent manner so that the generation can be executed just once offline before deployment.

1. For ease of description, we do not discuss communication latency here, which is a relatively independent problem from training latency, but we do incorporate communication latency into our sub-model allocation design and experimental evaluations.

During selector development, we identified a severe imbalance in training data label distribution and proposed two key improvements: first, through statistical analysis of rare label configurations, we shifted key configuration features toward smaller values and regenerated training datasets, effectively alleviating label distribution skew; second, we introduced focal loss function to dynamically adjust sample importance, ensuring the selector maintains high prediction accuracy even for rare layer algorithms.

Trained on comprehensive, high-quality data, the selector can accurately select the best algorithm to match the model configurations and runtime resources of the development chain. Specifically, our enhanced selector achieves 97% prediction accuracy, significantly outperforming traditional heuristic-based approaches that only achieve 73% accuracy. By further integrating the selected algorithm into our fine-grained training latency modeling, LATTE can accurately estimate the latency for each single-pass (forward and backward) propagation.

With accurate training latency estimation, LATTE facilitates configuring sub-models across devices to complete local training simultaneously. Due to our local training time estimation capability, we can move from traditional *server-driven assignment* to *client-side sub-model allocation*, which enhances adaptability to resource dynamics within and across training rounds. In addition, we also considered practical federated learning scenarios with dynamic device join and exit. By collecting device hardware information and historical resource information, we proposed two key designs, reliability aggregation and fast-β generator, successfully maintaining high performance and fast convergence in environments with dynamic device participation.

We develop LATTE as a middleware compatible to DL frameworks (*e.g.*, PyTorch and TensorFlow). It is lightweight, requires only a one-time offline building, and integrates transparently with mainstream FL frameworks like Flower [16], making it a readily deployable tool for building more robust and efficient real-world HFL systems. We evaluate LATTE on popular deep learning tasks by considering both IID and non-IID data distributions, including image classification on CIFAR-10/100 and OpenImage datasets, speech recognition using the Google Speech dataset, and human activity recognition using the HARBox dataset. We compare LATTE with seven classical or state-of-the-art methods, such as FedRolex [14] and TailorFL [24]. Extensive results show that LATTE not only significantly speeds up convergence across tasks by 1.22× to 3.18×, but also improves the accuracy of the central model by 2.49% to 9.79%. Moreover, LATTE exhibits unique orthogonal capabilities to personalized FL methods such as Hermes [38], demonstrating versatility in non-IID scenarios. Our main contributions are summarized below.

- We reveal the problem of development-chain diversity in federated learning systems and identify diverse layer algorithms as the key to explain the variability in training time. Based on this, accurate estimation of model training time can be achieved without complex operator or kernel-level modeling.
- We devise LATTE, with a novel layer algorithm selector and training time estimator, to accurately estimate the single-pass (forward/backward) propagation latency of

a model given its architecture, expected hardware and runtime memory. We further showcase its usability in a client-side sub-model selection for HFL.

- We conduct extensive experiments to evaluate LATTE in five typical HFL scenarios. The results show significant improvements in performance compared to seven classical or state-of-the-art methods.

2 Background & Motivation

We first review how system heterogeneity affects FL (§ 2.1) and then propose the definition of development chain diversity (§ 2.2), and finally point out the limitation of conventional training time modeling in FL (§ 2.3).

2.1 Primer of HFL

System Heterogeneity. Real-world FL deployments faces system heterogeneity [56]. One main reason is the diverse *computing capabilities* of participating devices. Such variations can lead to notable discrepancies in *training latency* of the same model, which affects the overall efficiency of FL.

In a typical FL process, such as the widely-used FedAvg [52], training occurs in *rounds*. Each device first downloads the current model from the server, trains the model on its local data for multiple epochs, and then uploads the updated model back to the server. Then the server aggregates these updates as the model for the next round. However, not all devices can complete the same amount of computation in each round. Slower devices can significantly delay the entire training process, leading to long wall-clock training time.

HFL Strategies. HFL is proposed to handle the system heterogeneity, and an effective approach studied in recent work [14], [25], [59], [24], [54], [48] is to tailor the model architecture to fit the capabilities of each device, *i.e.*, larger models are allocated to devices with higher capabilities, while smaller models are designated for less powerful devices. The goal is to synchronize the training completion time across all devices. These model-level operations lead to a general solution to cope with the system heterogeneity in FL systems, while we find that a key issue in their training time modeling that still limits the effectiveness of federation.[2]

2.2 Development Chain Diversity

In practical production environment that including different devices, there existing variations and inconsistencies during the runtime execution of machine learning models due to differences in the development tools, frameworks, and how they invoke the acceleration libraries, and we define this phenomenon as development chain diversity. Specifically, this concept acknowledges that even when the same neural network model is deployed on identical hardware, its performance can vary significantly rely on how development chain optimizing runtime processes. This runtime optimization often influenced by hardware specifications, layer configurations, and resource availability. In federated learning scenarios, this diversity becomes critical because edge devices may rely on different development chain, leading to invalidation of the existing training time modeling.

2. When the number of edge devices is quite large in FL systems, *e.g.*, hundreds or even thousands, some other orthogonal methods are also studied in the literature such as client selection and asychronous/semi-asychronous FL, which are detailed in related work (§5).

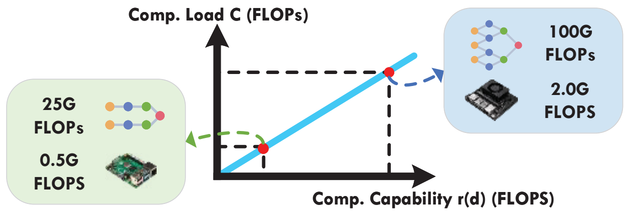


Fig. 1: Allocating a sub-model with computational load proportional to the capability r_d of device d.

2.3 Existing Training Time Modeling in FL

Most HFL studies [14], [25], [59], [24], [54], [48] focus on the allocation and aggregation of heterogeneous models for effective training. They match the models with devices by assuming a linear mapping between the network architecture and its training latency, expecting that the computational loads of the allocated models are within the devices' capacities and that all devices finish their training simultaneously. However, this method can lead to inconsistent training time estimates due to the *development-chain diversity*, as explained below.

Conventional Modeling. Assume local training uses standard mini-batch gradient descent. The training time T_{train} of a model architecture M on a device d can be estimated as:

$$T_{train} = E \cdot (b \cdot n) \cdot T, \tag{1}$$

where E is the number of epochs, b is the batch size, n is the number of batches, and T represents the latency of single-pass backpropagation (forward and backward) during training. Although E, b, and n are deterministic, latency T can vary from device to device. Specifically, the latency of a single pass of backpropagation is typically assumed to be proportional to the computational load C_M of the model M, *i.e.*,

$$T = \frac{C_M}{r_d}, \tag{2}$$

where r_d is a computing capability related coefficient, indicating the average processing speed of device d. For a fixed model architecture M, existing work assumes that C_M remains *constant*, suggesting that measuring the $r(d)$ suffices to accurately estimate the training latency [24], [25], which can be performed offline as a one-time effort, given known hardware specifications. We can then allocate sub-models to devices based on Equation (2), as shown in Figure 1.

Specifically, to ensure roughly the same training latency $T(M_1) = T(M_2)$, the server allocates sub-models M_1 and M_2 proportional to the capability of device d_1 and d_2, as follows:

$$\frac{C_{M_1}}{C_{M_2}} = \frac{r_{d_1}}{r_{d_2}}, \tag{3}$$

because the computational load C_{M_i} in Equation (3) of a model architecture M_i is constant given its topology. [3]

3. It should be noticed that this equation and following equation (4) represent ideal allocation principle.

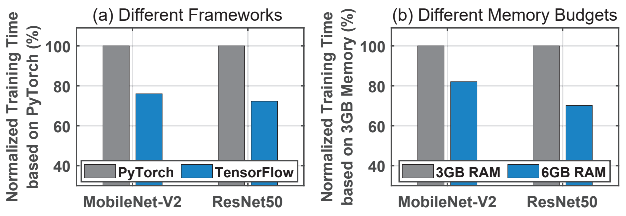


Fig. 2: Wall-clock training times of (a) using different frameworks on same models, and (b) using same framework on the same model but with different memory resources. Since absolute training times differ for each setting, we normalize them for clarity.

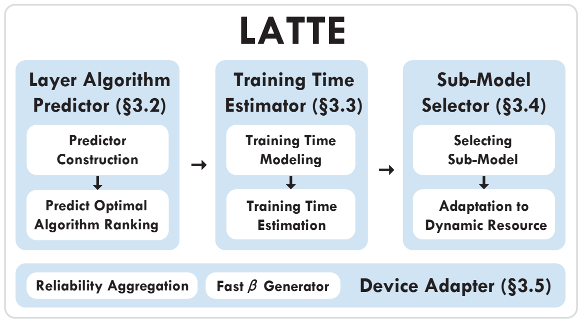


Fig. 3: Architecture of the LATTE design.

Limitations. However, this modeling of training time is oversimplified because computing capability is not the only source of system heterogeneity. Diversity in the development-chain of neural networks also matters. To reveal this, we conduct preliminary experiments on NVIDIA Jetson devices.

- We first run the same neural network on the same device, but using different deep learning frameworks. Figure 2(a) shows that across two popular models, the wall-clock training time of a single round with TensorFlow is only from 72.3% to 76.0% compared to using PyTorch.
- Even if developed using the same deep learning framework, models show different training times on the same hardware due to varied runtime resources (*e.g.*, memory). Figure 2(b) shows that the wall-clock training time of single epoch round with 6GB memory budget is only from 69.9% to 82.0% compared to a 3GB budget.

Implications. These experiments imply that training time is also related to the *runtime optimizations* of the deep learning framework. Furthermore, it links to *runtime resources* such as *memory*. These observations challenge the conventional training latency estimation and requires a new, more fine-grained modeling *beyond hardware specifications* to achieve more accurate training time estimation.

3 System Design

Our design is inspired by fine-grained analysis of the underlying runtime optimization strategy of current deep learning frameworks (§ 3.1), which reveals several key insights. Guided by the key insights, this section presents LATTE's design to solve the problems above. Figure 3 illustrates the functional modules of LATTE and here we list the detail functionality of each module.

- **Layer algorithm selector** (§ 3.2). Based on the key design insight to be described, this module first ranks candidate algorithms by considering the layer configuration, runtime memory, and hardware specifications. The selector then selects the fastest algorithm that satisfies the resource constraint as the output.
- **Training time estimator** (§ 3.3). This module further estimates the training time by explicitly incorporating the algorithm selection. We integrate the selected layer algorithm into our latency modeling and extend it to the entire model training. To achieve this estimation, we also propose effective ways to profile the relationship between computation and latency.
- **Sub-model allocator** (§ 3.4). This module finally employs the accurate training time estimation to allocate suitable sub-models for devices. It allows client-side sub-model adaptation to dynamic resource with a mechanism to ensure that all model parameters can be equally trained despite the varying sub-model architectures across different training rounds.
- **Dynamic Device Adapter** (§ 3.5). This module enhances LATTE's robustness to the dynamic joining and exiting of clients. It introduces a reliability aggregation mechanism to mitigate the impact of unstable clients and a fast-β generator to accelerate the onboarding of new clients, maintaining high performance and fast convergence in dynamic environments.

3.1 Key Observation and Insight

3.1.1 Diversity of Layer Algorithms

After we analyze the source code of current deep learning frameworks (*e.g.*, TensorFlow and PyTorch), we found that the same layer can have different implementations at runtime. For example, NVIDIA's cuDNN library include multiple algorithms (see Table 1) to implement forward convolution (`cudnnConvolutionFwdFilterAlgo_t`), *e.g.*, direct method (`CUDNN_CONVOLUTION_FWD_ALGO_DIRECT`), matrix product (`CUDNN_CONVOLUTION_FWD_ALGO_GEMM`), to name a few.

Different algorithms are also available for backward filter convolution (`cudnnConvolutionBwdFilterAlgo_t`), and also exist in (`cudnnConvolutionBwdDataAlgo_t`), which refers to backward data convolution, etc.[4] Devices that primarily use CPUs (*e.g.*, Raspberry Pi) also have diverse layer algorithms provided by their vendors [7], [1].

The same layer implemented by different algorithms results in different trade-offs. Some algorithms are general, which are compatible with most types of layer configurations, but their latency is always moderate, and their memory consumption is not small (*e.g.*, DIRECT in Table 1). In contrast, some algorithms are tailored for certain layer configurations, with optimized performance (*e.g.*, WINOGRAD in Table 1 is optimized for small kernels), but suffer

4. We use convolutions as an example to explain our insights and methods because they are among the most of optimized operations by deep learning frameworks and GPU vendors, which makes their latency estimation more challenging than less optimized operations.

Layer Algorithms	Generality	Memory Efficiency
GEMM	++	+
FFT	+	++
FFT_TILING	+	+++
IMPLICIT_GEMM	+++	++
IMPLICIT_PRECOMP_GEMM	++	++
DIRECT	++	+
WINOGRAD	+	+++
WINOGRAD_NONFUSED	++	++

TABLE 1: Candidate forward convolution layer algorithms and their characteristics. More '+' signs indicate better values for this characteristic category.

substantial latency when applied to relatively incompatible configurations. Due to this diversity, all frameworks require algorithm selection before model training. Although strategies in different frameworks can vary significantly, they generally follow two steps [12], [10]:

- **1) Ranking**: rank candidate algorithms for a given layer by the predicted latency based on their layer estimation strategies, and
- **2) Selecting**: estimate the corresponding memory footprint and select the top-ranked (fastest) algorithm that satisfies the predefined memory constraint.

Thus, the layer algorithm selection could affect the training time estimation in two aspects. First, given enough memory for the same model, layer algorithm selection strategies in different DL frameworks may produce different results, primarily because these strategies obtain rankings of the candidate algorithms in different ways (*e.g.*, heuristic strategy in PyTorch may not return the best ranking). Second, given limited memory budget, the same strategy may also return different algorithm ranking results even in the same device, since the selection must adhere to the memory limit. These two reasons lead to the obervations of Figure 2(a) and (b) in §2.

3.1.2 Reformulation of Training Time Modeling

The diversity of layer algorithms invalidates the conventional practice for sub-model allocation, *i.e.*, Equation (3), because different layer algorithms also lead to different latencies. Therefore, given a model architecture M, its computational load is a *function* of the layer algorithms $\{algo\}$ rather than a constant, *i.e.*, $C_M = C_M(algo)$. Hence, to ensure $T(M_1) = T(M_2)$ on two devices d_1 and d_2, the sub-models M_1 and M_2 should satisfy:

$$\frac{C_{M_1}(algo_1)}{C_{M_2}(algo_2)} = \frac{r_{d_1}}{r_{d_2}}, \tag{4}$$

where $C_{M_1}(algo_1)$ and $C_{M_2}(algo_2)$ are the actual workloads for M_1 and M_2 given layer algorithms $algo_1$ and $algo_2$. That is, the sub-model architecture allocated to the device should be *calibrated by layer algorithm*, as illustrated in Figure 4. Otherwise, training latency on devices with the same computing capabilities may differ significantly.

If we profile all layer algorithms in advance and consider their latency difference at each layer when allocating sub-models, devices can still complete per-round training simultaneously even when using different deep learning frameworks. However, beyond revealing this key design

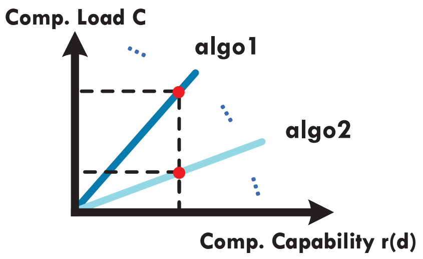


Fig. 4: The relationship between the model's computational load and the device's computing capability is essentially described by a series of lines due to the diversity of layer algorithms. Accurate training time estimation requires selecting the correct line first.

insight, we aim to go one step further in this paper: Can we pick the best (fastest) algorithm every time? If possible, we can use this strategy in different frameworks to ensure that each round of federated training can not only be completed synchronously across devices, but also the latency in each round can be minimized (to allocate the largest sub-model), which will further improve overall training efficiency and performance.

3.1.3 Design Challenge

However, it is difficult to achieve such best selection strategy for HFL systems. To understand this, we first analyze how layer algorithm selection is performed in existing deep learning frameworks, which usually form two categories: exhaustive testing (*e.g.*, `LaunchConv2DOp` in TensorFlow) and black-box heuristics (*e.g.*, `cudnnGetConvolutionForwardAlgorithm` in PyTorch).

The former, which exhaustively tests all layer algorithms on the target device and selects the fastest implementation, can achieve the best selection, but is very costly. Since sub-models could vary for each device and each training round in HFL systems, such high testing overhead may overwhelm the training latency. Figure 5 shows that even when all device memory is used, the latency of exhaustive testing (*e.g.*, TensorFlow) in first round training still accounts for 18.8–38.4% of the overall training latency. The latter uses heuristic rules to predict the best algorithm without actual on-device testing, which is fast but inaccurate, *e.g.*, its prediction accuracy is only around 73% of the exhaustive testing (see § 4.4.3), which can easily lead to inappropriate sub-model allocation.

Therefore, the main problem that challenges the design of new layer algorithm selection strategy is how to avoid the limitations of existing framework strategies and ensure the accuracy and efficiency of layer algorithm selection.

3.2 Layer Algorithm Selector

The above challenge motivates a new design of the layer algorithm selection without runtime assessments, which can override and replace the corresponding module in commercial deep learning frameworks to achieve a *lightweight* and *accurate* selection.

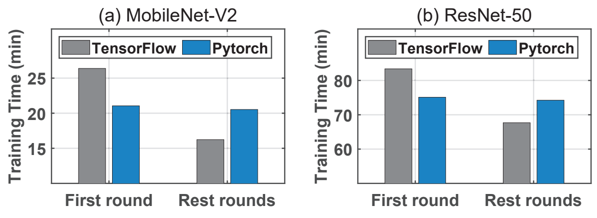
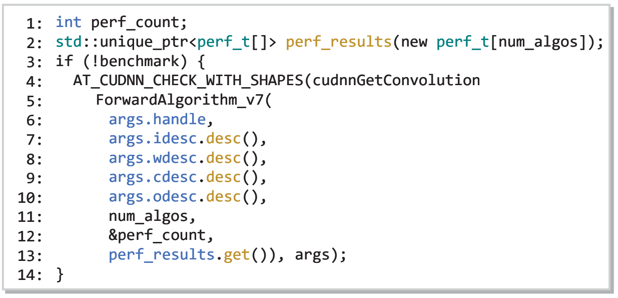


Fig. 5: In the first round of training, exhaustive testing strategy (*e.g.*, in TensorFlow) lead to much higher latencies than heuristic-based strategy (*e.g.*, in Pytorch) on two models (a) MobileNet and (b) ResNet.



```
int perf_count;
std::unique_ptr<perf_t[]> perf_results(new perf_t[num_algos]);
if (!benchmark) {
  AT_CUDNN_CHECK_WITH_SHAPES(cudnnGetConvolution
     ForwardAlgorithm_v7(
       args.handle,
       args.idesc.desc(),
       args.wdesc.desc(),
       args.cdesc.desc(),
       args.odesc.desc(),
       num_algos,
       &perf_count,
       perf_results.get()), args);
}
```

Fig. 7: Features of cuDNN's official heuristic API.

3.2.1 Idea and Design

Since all candidate algorithms have been given in the optimization library of each layer [12], [10], our main idea to achieve an accurate and practical design is to treat this problem as a classification problem and propose an efficient classifier for selection. We test different types of ML classifier like decision trees, SVMs and XGBoost but found their low accuracy as heuristic rules. Fortunately, we found the MLPs can meet these requirements effectively, like lightweight and high accuracy. The testing results are show in Figure 6. Below, we introduce our MLP selector design.

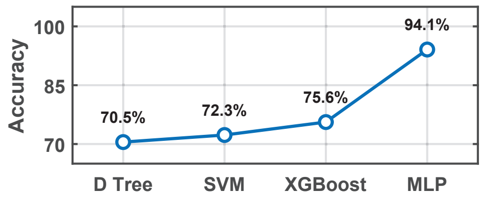


Fig. 6: Performance of different ML classifiers.

1) Architecture of the selector. For lightweight selection, we design a simple multi-layer perceptron (MLP) classifier. It takes a layer configuration as input and outputs a score for each layer algorithm, indicating its latency ranking. We reuse the inputs of existing heuristic strategy's API (*e.g.*, I/O dimension, convolution kernel size, padding, stride, dilation *etc.* in `cudnnGetConvolutionForwardAlgorithm` as inputs in our MLP, see Figure 7). The output of the softmax layer represents the probability of the fastest layer algorithm (and thus their rankings). We design a 6-layer MLP from the result of Neural Architecture Search with speed and accuracy target and Cross Entropy due to it is the most commonly used loss function for classification task.

2) Feature enhancement. We further integrate feature importance analysis method [31] to pick important features, into our design to improve performance. For example, `stride_height` and `stride_width` tend to be more important than `input_h`, `input_w`, and `output_c` for forward convolution. Thus, we add an attention layer after the input layer to assign greater weights to these important features.

3.2.2 Training Dataset Construction

With the above classifier design, we further propose a tool, responsible to acquire and label diverse layer configurations to train the classifier.

1) Layer configuration generation. The acquisition tool first samples and then derives a wide spectrum of layer configurations, including I/O size, filter size, stride, padding, *etc.*, from a set of models widely used in HFL scenarios. In our current development, the tool samples from 28 widely used models (see Table 2), resulting in 30,000 configurations.

Model Family	Model Instances
VGG	VGG-11/13/16/19
MobileNet	MobileNet-V1/V2
ShuffleNet	ShuffleNet-V1/V2
ResNet	ResNet-18/34/50/50-V2/101/101-V2/152/152-V2
EfficientNet	EfficientNet-B0/B1/B2/B3/B4/B5/B6/B7
Others	GoogLeNet/SqueezeNet/Xception/DenseNet-121

TABLE 2: Models used to collect layer configurations.

2) Ground-truth acquisition. After collecting sufficient configuration samples, the main issue is to obtain the ground-truth, *i.e.*, which algorithm is the best choice for each configuration that is difficult to analyze manually. To solve this issue, we propose to reuse the functional APIs of the exhaustive testing-based deep learning frameworks in our design. However, current APIs are binding with deep learning frameworks, we cannot utilize them independently.

To address the above issue, we proposed that by separating the exhaustive search module from the framework, the module can be re-engineered into a lightweight, standalone groundtruth acquisition tool, and the tool thus can be invoked independently without training. Specifically, we referenced the implementation of the exhaustive search mechanism within the TensorFlow source code. We then manually re-engineered this functionality by creating the necessary data structures and descriptors (e.g., `cudnnTensorDescriptor_t`, `cudnnFilterDescriptor_t`, `cudnnConvolutionDescriptor_t`).

While this initial separation requires careful manual coding, the resulting ground-truth acquisition tool is fully automated and general. It is not limited to a specific layer type. It can be invoked independently to find the best algorithm for any layer that has multiple candidates, simply by providing the appropriate layer configurations. Once this tool is built, it can generate ground-truth data for tens of thousands of configurations without any further manual intervention.

3) Obtaining Ground-truth. By using the tool, now we can easily obtained the groundtruth without redundant operations. As discussed previously (§3.1), this type of exhaustive

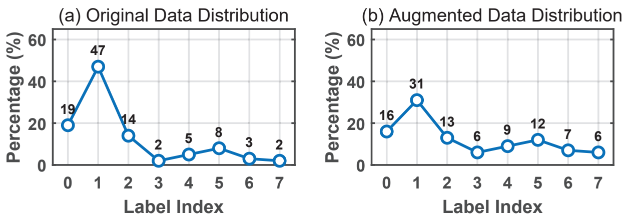


Fig. 8: (a) Original data distribution. (b) Augmented data distribution.

search strategy is very slow during on-device tests. However, since the latency ranking is based on the computational load of each algorithm rather than the absolute latency value, the ground-truth data is device-independent, and the overhead of collecting it is acceptable when it occurs during a one-time offline phase on a powerful server. Furthermore, we consider distributing the obtained layer configurations to clients with GPU capabilities in proportion to their computational power. This enables parallel processing between the server and clients, reducing the time overhead.

After training with layer configurations and their corresponding best choices collected by the tool, we can obtain an accurate selector that is also lightweight and does not require any runtime testing when selecting layer algorithms.

Incorporating Memory Constraint. Given the algorithm rankings obtained by the classifier, the selector ultimately selects the fastest algorithm within memory budget mem as output (to be used by the training time estimator). We employ an accurate estimator (`cudnnGetConvolutionWorkspaceSize`) to retrieve the memory usage of each layer algorithm.

3.2.3 Enhancing the Layer Algorithm Selector Performance

While our selector significantly outperforms heuristic strategies (e.g., 94% vs. 73% accuracy), a performance gap remains compared to an exhaustive search. We identified the primary cause as a severe imbalance in our training dataset.

1) Skewed-Dataset Problem. Statistical analysis revealed that three algorithms (labels 0, 1, and 2) accounted for approximately 80% of our training samples, as shown in Figure 8(a). This skew reflects how deep learning frameworks often default to these algorithms in practice. However, this imbalance impairs the selector's ability to generalize, leading to poor performance on rare labels that correspond to critical, specialized use cases.

2) Data Augmentation. To address this, we first attempted data augmentation. We observed that the most frequent labels were typically associated with configurations having larger input, kernel, and padding sizes. Based on this, we generated new layer configurations using smaller values for these key factors to specifically augment the rare labels. This strategy improved the label distribution (see Figure 8(b)) but only partially mitigated the overall imbalance.

3) Limitation of current loss function. Since data augmentation alone was insufficient, we turned to improving the MLP's training components. The standard Cross-Entropy loss function, $\text{CE}(p_t) = -\log(p_t)$, is ill-suited for imbalanced datasets because it treats all classes equally. This causes the model to prioritize accuracy on the dominant classes, while the minimal loss contribution from rare classes leads to poor performance on them.

4) Focal Loss. We first conducted multiple re-sampling experiments but still found significant class imbalance, because some algorithms are actually rarely selected in practice, making re-sampling method ineffective.

Among the potential solutions, focal loss [46] offers a promising solution to class imbalance by down-weighting well-classified examples and focusing on difficult or under-represented samples. Its formula is:

$$\text{FL}(p_t) = -\alpha_t \cdot (1 - p_t)^\gamma \cdot \log(p_t) \tag{5}$$

where p_t is the model's predicted probability for the true class, α_t is a class-balancing factor, and γ is the focusing parameter that reduces the contribution of well-classified examples.

For our MLP training, we set $\gamma = 2$ (the default value effective for most imbalanced datasets) and made α_t inversely proportional to class frequency ($\alpha_t = \frac{1}{class_{t\_freq}}$). This approach assigns higher weights to rare classes and lower weights to common ones. Implementation of these parameters successfully improved model accuracy.

3.3 Training Time Estimator

The selector (§3.2) is designed to accommodate the diversity of layer algorithms to achieve accurate training time estimation. In this subsection, we describe how to integrate the selection results into our training time estimator.

In general, given a model M, a memory budget mem, and a target device d, the training time estimator (see Figure 9(a)) predicts its single-pass propagation latency T. We decompose the model into *key* layers and *non-key* layers (§ 3.3.1), integrate layer algorithms into the latency modeling of key layers (§ 3.3.2), and profile the coefficients for both key and non-key layers (§ 3.3.3). This approach can provide accurate latency estimates with low profiling overhead.

3.3.1 Key Layer Identification

Given a model, its key layers are the layers with more than one layer algorithms defined in the deep learning framework, *e.g.*, for CNNs, these mainly refer to the convolutional layers, while those with a single and same algorithm implementation are defined as non-key layers, *e.g.*, FC and activation layers. Therefore, we can easily classify each layer as key or non-key based on layer type.

3.3.2 Layer Algorithm Aware Training Time Modeling

We incorporate the layer algorithms by extending traditional latency modeling *i.e.*, Equation (2), as follows:

$$T(d) = \left(\sum_i \frac{C^{key}_{fwd}(algo_i)}{r^{key}_{fwd}(d)} + \sum_i \frac{C^{key}_{bwd}(algo_i)}{r^{key}_{bwd}(d)} \right) + \left(\frac{C^{non}_{fwd}}{r^{non}_{fwd}(d)} + \frac{C^{non}_{bwd}}{r^{non}_{bwd}(d)} \right), \tag{6}$$

where $C^{key}_{fwd}(algo_i)$ and $C^{key}_{bwd}(algo_i)$ are the computational loads of key layer i of model M in the forward and backward pass, implemented by layer algorithm $algo_i$. Similar

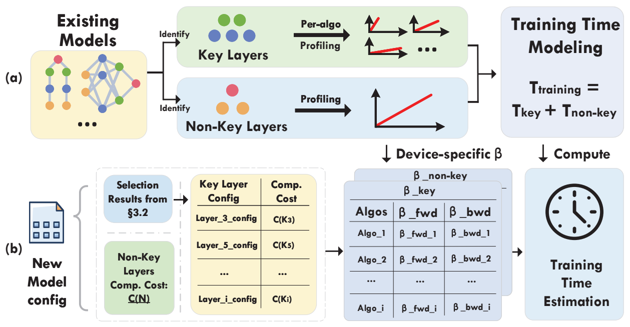


Fig. 9: (a) Construction of training time estimator. (b) Training time estimation for a new model.

to existing deep learning frameworks, LATTE also selects the same algorithm for the same key layers of forward and backward propagation. Similarly, C_{fwd}^{non} and C_{bwd}^{non} are the *total* computation of all the non-key layers of model M in the forward and backward pass. The latency is linear to the computational load when the layer algorithm can be determined, given coefficients $r_{fwd}^{key}(d)$, $r_{bwd}^{key}(d)$, $r_{fwd}^{non}(d)$, and $r_{bwd}^{non}(d)$. The rationale of our training latency modeling is as follows:

- We decompose the model M into key and non-key layers to reduce the profiling overhead while maintaining high latency estimation accuracy.
- As non-key layers have a single and same implementation, they are modeled the same as the traditional approach, *i.e.*, Equation (2), and can be profiled in advanced and applied in forward and backward passes.
- The modeling of key layers explicitly takes into account the diversity of layer algorithms.

For practical usage, we further rephrase Equation (6) as:

$$T(d) = \sum_i (\beta_{fwd}^{key}(algo_i, d) + \beta_{bwd}^{key}(algo_i, d)) \cdot C(K_i) \quad (7)$$
$$+ (\beta_{fwd}^{non}(d) + \beta_{bwd}^{non}(d)) \cdot C(N),$$

where $C(K_i)$ is the computational load of key layer K_i with direct implementation (*e.g.*, the direct method for convolution), and $C(N)$ is the total computations loads of all non-key layers. Both $C(K_i)$ and $C(N)$ are deterministic given the model architecture M. In contrast, $\beta(algo, d) = C(algo)/r(d)$ absorbs all algorithm- and device-dependent variations. This formula makes it easy to estimate the single-pass training latency for each layer from a table (see Figure 9(b)).

It should be noticed that due to optimizations of modern operating systems and hardware (e.g., efficient memory bandwidth management [79] and caching strategies [22]) significantly mitigate the effects of non-linear interactions (*e.g.*, memory bandwidth bottlenecks, cache effects) in most practical deployment scenarios, making our linear assumption reasonable and effective under most conditions. However, in some extreme conditions, non-linear interactions may lead to inaccurate estimations. We will undertake this challenge in the future.

3.3.3 *Profiling Coefficient β*

From Equation (7), we should profile $\beta_{fwd}^{key}(algo, d)$, $\beta_{bwd}^{key}(algo, d)$, $\beta_{fwd}^{non}(d)$ and $\beta_{bwd}^{non}(d)$ in advance for runtime training latency estimation. The profiling is performed for all device types in the federation. Furthermore, we need to profile each layer algorithm for each key layer $\beta_{fwd}^{key}(algo, d)$ and $\beta_{bwd}^{key}(algo, d)$. Recall that $\beta(algo, d) = C(algo)/r(d)$ without coupling between $algo$ and d. So, the profiling overhead scales linearly with $\max\{algo, d\}$ rather than their product, which is more manageable. We reuse the model family in Table 2 and extract $1,000$ layer configurations to profile the coefficients in the current design of LATTE. We manage β by periodically monitoring processor utilization and updating β by multiplying the new utilization to avoid the overhead of re-estimating it. It should be noticed that LATTE monitors device temperature before profiling, ensuring a stable thermal state by cooling the device to normal operating temperatures, while employing short profiling sessions to prevent overheating; additionally, it terminates or pauses non-essential background processes and monitors CPU, GPU, and memory usage in real-time to prevent resource contention. Upon detecting thermal throttling or abnormal resource usage, LATTE pauses profiling until the device cools down or resources stabilize.

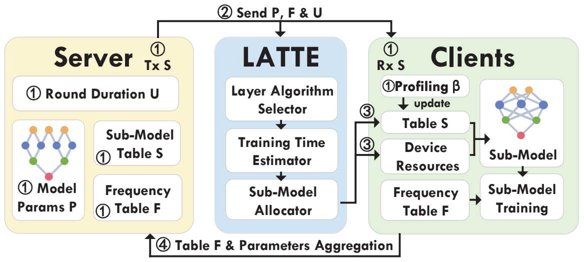


Fig. 10: Overall HFL workflow with LATTE.

3.4 Sub-Model Allocator

With accurate training latency estimation, we describe how LATTE allocates sub-models to devices to ensure synchronized local model training in each round for HFL systems.

Unlike the previous HFL schemes [14], [25], where clients *passively* train server-assigned sub-models, clients in LATTE can *actively* determine sub-models to better suit their own resource dynamics due to LATTE's local training latency estimation capabilities. In the following, we first describe the entire client-server interaction in LATTE (§ 3.4.1) and then detail our client-side sub-model allocation design (§ 3.4.2).

3.4.1 *Overall Client-Server Interactions*

Figure 10 shows the overall HFL workflow with LATTE.

① On initialization, the server determines the round duration U, the global model architecture M, its complete model parameters P, and the frequency table F (introduced and used in §3.4.2). The server then generates candidate sub-model configurations (200 configurations in our case, which is a trade-off between the granularity of sub-model assignment and the estimation overhead, *e.g.*, it takes about 40 seconds to estimate 200 sub-models on a Jetson Nano device, and we will automate the selection of this number in the future.) via model scaling, storing them into the sub-model table S. Note that other methods of generating

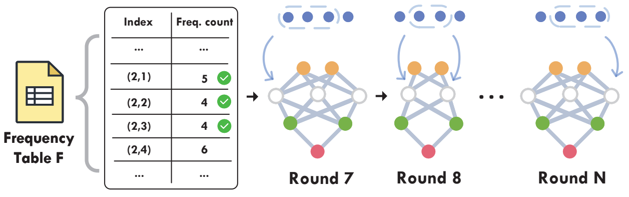


Fig. 11: Parameter prioritizing via frequency table F.

candidate architectures are also applicable. Afterwards, the server sends the sub-model configuration table S for all clients. Each client profiles its layer algorithm- and device-dependent coefficients β, then updates the sub-model table S, estimates its training latency and measures its memory usage. These estimation can be performed quickly and are a one-time effort.

② In each round, the server sends the current complete model parameters P, round duration U, and frequency table F to all clients for local training.

③ Each client polls the current memory budget mem and bandwidth B via the bandwidth monitoring tool *e.g.*, bmon [2]. It then calculates the overall training time T_{train} using the single-pass training latency T stored in the sub-model S and the communication latency T_{comm} based on the bandwidth B and the model size [24]. Each client then selects the largest sub-model configuration in S such that $T_{train} + T_{comm} \leq U$, instantiates the sub-model from the full model parameters P and updates the frequency table F, following the scheme in § 3.4.2, and then starts training on its local dataset.

④ The local parameters and frequency table F are sent back to the server for aggregation. We apply the standard aggregation scheme as in FedAvg [52]. Since the client submits a sub-model to the server, only the updated parameters are averaged [25], [14], [82], [24]. Due to accurate training time estimates, all devices are expected to return updated model parameters within the deadline U. Steps ②-④ are iterated until the global model converges.

3.4.2 *Sub-Model Allocation at Client*

As mentioned above, LATTE allows dynamic client-side sub-model allocation to adapt to resource dynamics. However, such adaptation may affect the effectiveness of federated training. This is because if sub-model allocation takes resource dynamics into account, parameters in the global model may not be equally trained. In LATTE, we design a *priority-aware sliding window* scheme for sub-model allocation, which facilitates efficient training while maintaining adaptability to resource dynamics.

Our main idea is to train a subset of model parameters in rounds on a rolling basis, so that all parameters are trained evenly in the long run. Previous work [14] implemented this idea, and it was also shown to converge [84]. This approach uses a sliding window for each layer of fixed size and stride length, assuming that the sub-model size is the same across rounds, but this is invalid in our problem as the sub-model size on the client may change due to available resources.

Method. Since LATTE selects a different sub-model architecture in each round based on current resource availability to meet local training deadlines, the sub-model architecture naturally translates into a set of windows per layer, the size of which can vary from round to round. Therefore, the challenge is to ensure even training of all parameters under different window sizes over rounds. Our solution is to *prioritize* rarely trained model parameters into a sliding window for training. This is achieved by tracking how often the model parameters are trained. Specifically, we maintain a parameter trained frequency table F on each device, recording the training frequency of each parameter. The server also maintains a general table F_{server}. Only when the sub-model architecture changes due to resource dynamics, the client will upload its parameter trained frequency table to the server for aggregation and then distribute to each client. The size of each table is small, *e.g.*, only 2.3–24.5 MB, and the communication overhead of maintaining these tables is negligible. Figure 11 shows our parameter prioritizing process across rounds.

3.5 Dynamic Device Adapter

In practical federated learning scenarios, dynamic device participating and exiting is common. However, existing LATTE designs do not take this factor into account, resulting in some problems. For example, when existing devices exit, it will affect the final model's convergence and performance, while new devices participating require device factor profiling from scratch, which is a time-consuming process. To address these, we propose dynamic device adapter in this chapter as shown in Figure 12, maintaining two tables on server and two corresponding design, Reliability Aggregation (green part of Figure 12) and Fast-β Generator (blue part of Figure 12) to support efficient training in scenarios with dynamic device participating and exiting.

3.5.1 *Device Information Collector*

At first, we maintaining two tables in server side, specifically:

- A table A_{rs} of device real-time resource, including available memory, bandwidth condition and CPU/GPU utilization. After federated learning begins, clients send resource status to the server each round (less than 1 KB). When server does not receive updates from device, it records the device as exit status.
- A table A_{hw} of device hardware information, including hardware specifications (*e.g.*, CPU/GPU architecture, core frequency) and β coefficients. Before federated learning begins, clients register their hardware specifications and β coefficients with profiling records to the server.

These two tables are maintained for next two section (2) and (3), which will describe in detail later.

3.5.2 *Reliability Aggregation*

The reliability aggregation is a server-side module (green part of Figure 12) that enhances model aggregation robustness through reliability assessment and weighted aggregation, to relief the effect of device exiting. Specifically, by utilizing the historical resource data trace provided by the table A_{rs} in server, we set a reliability score to represent the reduction of available resources in the short term in each device.

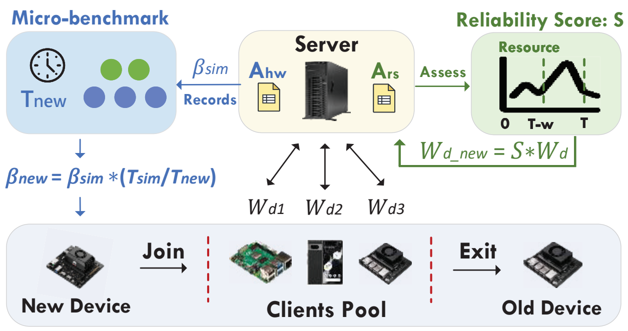


Fig. 12: Overview of design dynamic device adapter.

In detail, we define a sliding window W_s, which used to analyze the short term resource dynamics provided by A_{rs}. For the device D and resource type R_i, the available resource value of the rounds T are denoted as $R_{(d,i,t)}$. Then we determine whether the resources have decreased significantly by comparing with adjacent rounds. We define the tolerance threshold ϵ_r. If $R_{(d,i,t)} \geq (1-\epsilon_r) * R_{(d,i,t-1)}$, it is considered that the resources have not decreased significantly, indicating that the equipment is keeping stable in this round. In W_s round, count the stable round ratio as reliability score S_i of resource type R_i. We further considering multiple resources, the total reliability score S of device D is the average of the scores of each resource S_i. In this way, the influence of different resources can be balanced to ensure a comprehensive assessment of equipment reliability.

The reliability score is used for weighted model update. For each device D that submits update, the original weight to be uploaded is W_d. By integrating our design, the weight to be uploaded become $W_{d\_new} = S * W_d$. When S=1, all resources do not show a significant decrease in a short time, refer to the stable and reliable, the uploaded weight remains unchanged, and the default aggregation method will be executed subsequently. When S<1, there is a significant resources reduction in short time, indicating a decrease in the reliability of this device. Therefore, the importance of this device should be reduced. This method ensures that devices with stable resources contribute more to the global model and reduces the impact of unreliable devices.

It should be noticed that the tolerance threshold ϵ_r is the important hyperparameter, initially initialized at 10%. As federated learning progresses, the server continuously collects the resource variation amplitude of the actual exiting devices, and then updates this hyperparameter by taking the typical fluctuation amplitude.

3.5.3 Fast-β Generator

The fast-β generator is a client and server coordinate module (blue part of Figure 12) designed to accelerate new devices participating into FL by quickly estimating β coefficients. Specifically, when new devices are joined, by querying the table A_{hw} in server, the most similar devices are identified based on hardware specifications (such as CPU/GPU architecture and core frequency), then their device factors β_{sim} and profiling records are sent to new device.

Subsequently, the new device performs lightweight micro-benchmark profiling (sampling around 20 layers in original 1000 layers profiling) in each algorithms, and measures the execution time $T_{(\text{new},l)}$. By comparing with the results of similar devices $T_{(\text{sim},l)}$ (from profiling records), scaling factors are calculated as $S = \frac{T_{(\text{sim},l)}}{T_{(\text{new},l)}}$, thereby estimating the β values for new device $\beta_{\text{new}} = \beta_{\text{sim}} \times S_l$.

Finally, the new device uploads hardware information and estimated β values with profiling records to the A_{hw} in server for future use. This method greatly reduces the initialization time from the minutes-level device factor profiling to the seconds-level micro-benchmark, accelerating the new devices join the FL.

In conclusion, the Dynamic Device Adapter is a modular enhancement for HFL's practical robustness, with two components having different dependencies on our core time estimator. The first, a Reliability Aggregation Mechanism for handling device exits, is general in most of scenarios. It uses historical data to assign reliability weights to clients, a feature broadly applicable to any weighted aggregation scheme like FedAvg. The second, a Fast-β Generator for new devices, is tailored for LATTE. It specifically accelerates the initial profiling for our time estimator, serving as an input to enable fast and accurate predictions for new clients. While the adapter enhances practicality, our paper's central contribution is the time estimator itself, which stands on its own.

4 Evaluation

4.1 Implementation

We implement LATTE using Python 3.6, C++ 14, CUDA 10.2 [4], PyTorch 1.6.0 [9] and TensorFlow 1.14.0. [11]. To train the selector, we generate large amount of key layer configurations and then conduct benchmarking to obtain the ground truth with our acquisition tool. The development of acquisition tool is supported by cuDNN [5] and is compiled using NVCC [8]. Then the selector function overrides the default layer algorithm selection function [10], [12] in source code of deep learning frameworks (e.g., in the file `aten/src/ATen/native/cudnn/Conv_v7.cpp`) and then recompiling the frameworks, which ensures native, overhead-free execution without altering the user-facing API. For practical HFL scenarios and evaluation, LATTE is integrated into the advanced and mainstream FL framework Flower [16].

4.2 Experimental Setups

4.2.1 Test-bed

We evaluate on a client pool of 10 devices, consisting of three tiers as shown in Table 3. We include 2 devices per type, running different frameworks, *i.e.*, TensorFlow for one and PyTorch for the other. The central server is a computer equipped with Intel i7-9700K CPU and NVIDIA RTX 2080Ti GPU. Default bandwidth is 100Mbps and we use Wondershaper [13] to control the bandwidth of each client. Device processor frequency is fixed during evaluation.

4.2.2 Tasks, Datasets, and ML Models

We test on IID datasets and real-world non-IID datasets. Each dataset corresponds to a task, and we test different models for each task.

Category	Type	RAM	CPU	GPU
Higher-end	Dell Inspiron 5577	8GB	i5-7300HQ	GTX 1050
Mid-tier	Jetson Xavier NX Jetson TX2	8GB 8GB	ARMv8.2 ARM A57	Pascal Volta
Lower-end	Jetson Nano Raspberry Pi 4B	4GB 8GB	ARM A57 ARM A72	Maxwell —

TABLE 3: Device configuration in the client pool.

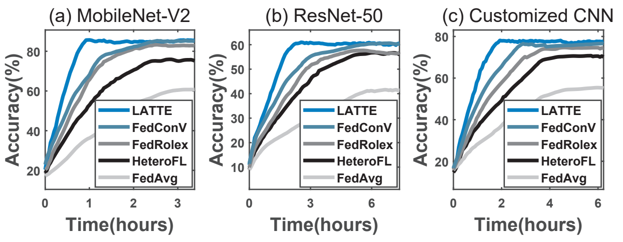


Fig. 13: Overall performance on IID datasets.

i) Tasks with IID Datasets.

- **Image Classification.** We use CIFAR-10 and CIFAR-100 [3] and train MobileNet-V2 [60] and ResNet-50 [27] respectively for these tasks.
- **Human Activity Recognition.** We utilize the classical HAR [15] dataset, and train a simple customized CNN with 3 convolutional layers and 2 dense layers.

ii) Tasks with non-IID Datasets.

- **Image Classification.** We utilize the OpenImage [36] dataset, where we select 50 image classes, and train MobileNet-V2 for this task.
- **Speech Recognition.** We choose the Google Speech dataset [71], and train ResNet-50 for this task.
- **Human Activity Recognition.** We choose the HAR-Box [55] dataset and adopt the same pre-processing method as [40]. We train the same customized CNN as in i) the IID scenario for this task.

4.2.3 Baselines

We compare LATTE with the following:

- **HeteroFL [25]**: a classic HFL method that assigns different sub-models to devices according to their computing capabilities.
- **FedConv [61]**: a state-of-the-art HFL scheme with learning the parameters of the heterogeneous sub-models via convolutional compression to improve training accuracy.
- **FedRolex [14]**: a state-of-the-art HFL scheme with a sliding window sub-model allocation mechanism to improve the training accuracy.
- **FedAvg [52]**: the classic generic FL algorithm without accounting for system heterogeneity.
- **TailorFL [24]**: the state-of-the-art FL method which considers both data and system heterogeneity.
- **Fjord [29]**: a classical dropout based HFL method to handle data heterogeneity.
- **FedSEA [63]**: the state-of-the-art semi-asynchronous FL method for edge devices.
- **FedAsync [74]**: a classical asynchronous FL method.
- **Oort [37]**: a classical client selection based HFL method to handle system heterogeneity.

4.2.4 Evaluation Metrics

We compare different methods using the following metrics.

- **Time-to-Convergence.** It is the actual wall-clock time for training a model till convergence. Not all methods will converge within a specified time limit.
- **Model Test Accuracy.** It is the accuracy on the test datasets obtained by the model trained through FL.
- **Estimator Precision.** It is the precision between the estimated latency and the actual training time, *i.e.*,

$$Estimate\ Precision = 1 - \left|\frac{T_{estimate} - T_{real}}{T_{real}}\right|, \quad (8)$$

where $T_{estimate}$ is from the training time estimator (§ 3.3) of LATTE, and T_{real} is the real wall-clock training time. A high precision means an accurate estimator.

4.3 Overall Performance

4.3.1 Performance on IID Datasets

This experiment compares LATTE with the-state-of-the-art HFL schemes to handle system heterogeneity alone.

Setups. We compare LATTE with HeteroFL [25], FedRolex [14] and FedConv [61], three representative HFL schemes that allocate diverse sub-models to devices according to their capabilities. To isolate the impact of system heterogeneity, we conduct experiments on three IID datasets (CIFAR-10, CIFAR-100, and HAR). We also include FedAvg [52] as the baseline that does not explicitly address system heterogeneity.

Results. Figure 13 shows the results on the three IID datasets.

Time-to-Convergence. LATTE consistently achieves the fastest convergence. When training MobileNet on CIFAR-10, LATTE converges in 1.03 hours, 2.14×, 2.72× and 3.18× faster than FedConv, FedRolex and HeteroFL, respectively. FedAvg fails to converge within the 20-hour limit, highlighting the necessity to explicitly deal with system heterogeneity. When training ResNet-50 on CIFAR-100, LATTE converges in 2.60 hours, shortening the convergence by 1.40×, 2.36× and 2.57× than FedConv, FedRolex and HeteroFL, respectively. LATTE's time-to-convergence is 1.98 hours when training Customized CNN on HAR, 2.18×, 2.89× and 3.03× faster than FedConv, FedRolex and HeteroFL, respectively.

Model Test Accuracy. LATTE also achieves the highest test accuracy at convergence, reaching an accuracy of 85.78%, 60.99%, and 78.11% on MobileNet, ResNet, and Customized CNN respectively, which is 2.49% to 3.45% higher than FedRolex, and 4.23% to 9.79% higher than HeteroFL. FedConv performs comparably to LATTE, as FedConv mainly optimize the model accuracy.

Figure 13 shows that LATTE notably improves both the efficiency and effectiveness of federated learning under system heterogeneity than the state-of-the-art HFL methods. It should be noticed that performance differences across tasks are common and often stem from task-specific characteristics, such as dataset size, model complexity, and data

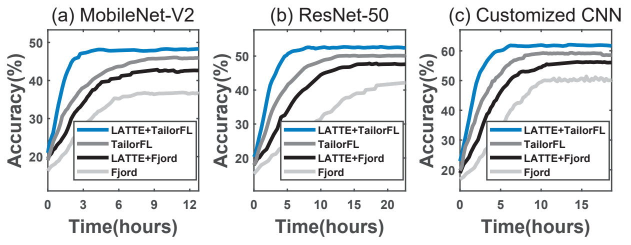


Fig. 14: Overall performance on non-IID Datasets.

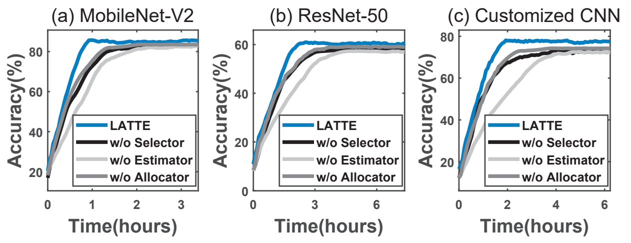


Fig. 15: Contributions of individual modules.

distribution. Therefore, the generalizability of LATTE is guaranteed.

4.3.2 Performance on non-IID Datasets

This experiment aims to further evaluate the effectiveness of LATTE under both data and system heterogeneity. As a transparent design, LATTE can be easily integrated with FL solutions that handle data heterogeneity to improve their efficiency.

Setups. We integrate LATTE into Fjord [29] and TailorFL [24], two representative FL schemes that combat data heterogeneity by training personalized models. We test their performance on three non-IID datasets (OpenImage, Google Speech, and HARBox) with heterogeneous edge devices with and w/o the enhancement of LATTE.

Results. Figure 14 shows performance gains with LATTE.

Time-to-Convergence. The personalized FL methods converges notably faster after adding LATTE. When training MobileNet on OpenImage, the LATTE-enhanced TailorFL and Fjord converge in 3.91 hours and 5.92 hours respectively, which are $2.72\times$ and $1.24\times$ faster than the standalone versions. When training ResNet-50 on Google Speech, LATTE +TailorFL converges $2.09\times$ faster than TailorFL, and LATTE +Fjord is $1.29\times$ faster than Fjord. When training Customized CNN on HARBox, TailorFL and Fjord are accelerated by $2.42\times$ and $1.26\times$, respectively.

Model Test Accuracy. Integration with LATTE also improves the test accuracy. LATTE improves the accuracy of TailorFL by up to 2.81%, reaching 48.29%, 52.86%, and 62.14% on MobileNet, ResNet, and Customized CNN, respectively. Similarly, LATTE improves the accuracy of Fjord by up to 5.82%, reaching 42.82%, 47.81%, and 56.22% on MobileNet, ResNet, and Customized CNN, respectively.

These results affirm that LATTE can complement existing FL methods for data heterogeneity, achieving both accurate and fast training under both data and system heterogeneity.

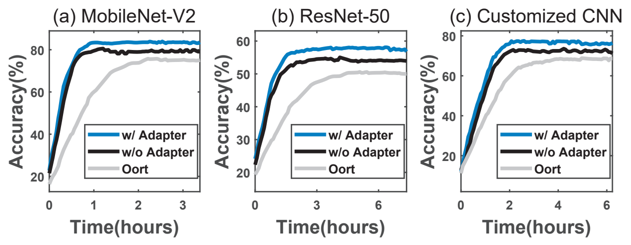


Fig. 16: Performance of Dynamic Device Adapter.

4.4 Ablation Study

4.4.1 Performance Breakdown

This experiment evaluates the contributions of each module to the overall performance.

Setups. We implement three variants of LATTE:

- **LATTE w/o Selector:** it replaces the layer algorithm selector (§ 3.2) with the default APIs [12], [10] provided by deep learning frameworks.
- **LATTE w/o Estimator:** it replaces the training time estimator (§ 3.3) with the naive modeling in Equation (2).
- **LATTE w/o Allocator:** it replaces the sub-model allocator (§ 3.4) with the sliding window scheme in [14].

Results. Figure 15 shows the results of different variants.

Time-to-Convergence. The variant w/o selector suffers from notable slowdown ($0.46\times$ to $0.53\times$ that of LATTE). This is because the default APIs may not recommend the fastest layer algorithm, leading to inaccurate time estimates. The variant w/o estimator also experiences significant slowdown ($0.34\times$ to $0.54\times$ that of LATTE). This is because the naive latency modeling ignores the impact of layer algorithms, which is the key to inconsistent training time estimates. The variant w/o selector shows a mild slowdown ($0.56\times$ to $0.65\times$ that of LATTE), mainly due to the lack of adaptation to resource dynamics. This can lead to sub-model mismatch when resources on the device change frequently.

Model Test Accuracy. Overall, the degradation in test accuracy is less severe than the degradation in convergence speed. On the three datasets, for the variants w/o selector, estimator, and allocator, the accuracy drops by 2.68% to 4.78%, 2.97% to 5.44%, and 1.69% to 3.81%, respectively.

In summary, removing individual modules from LATTE mainly affects the convergence speed. The convergence slowdown is less drastic for the variants w/o selector and w/o allocator than that w/o estimator. This is because the default APIs still provide about 73% layer algorithm prediction accuracy, and system resources do not vary all the time. In contrast, the variant w/o estimator falls back to the HFL scheme ignoring the diversity of layer algorithms, which is the primary motivation of our work.

4.4.2 Effectiveness of Dynamic Device Adapter

Setups. Specifically, we set up a dynamic clients pool, where 1-3 clients randomly join or leave at regular intervals. We also selected the client selection method Oort and LATTE without adapter as baselines.

Results. The experimental results show that compared to traditional client selection algorithms and LATTE without

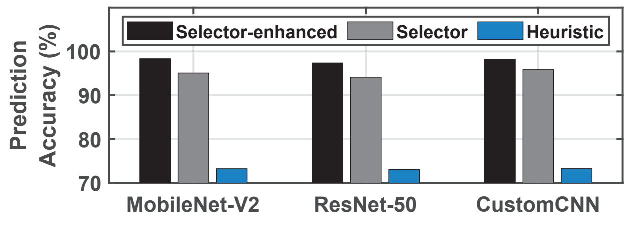


Fig. 17: Effectiveness of the layer algorithm selector.

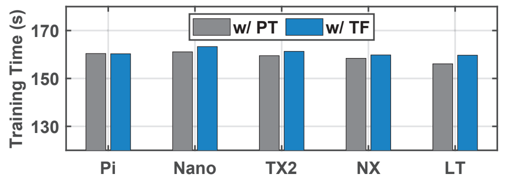


Fig. 19: LATTE alleviates the variations in training time across deep learning frameworks.

consideration for dynamicity, LATTE with the dynamic device adapter achieved faster model convergence and higher accuracy in dynamic environments, exceeding from 3.06% to 8.65% final accuracy and converge from $1.26\times$ to $2.56\times$ faster respectively. This demonstrates that the introduced dynamic device adapter provides robustness to handle devices dynamically joining and exiting.

4.4.3 *Effectiveness of Layer Algorithm Selector*

A key to the convergence speedup of LATTE is the layer algorithm selector (§ 3.2). This experiment zooms into its performance against the default APIs in deep learning frameworks.

Setups. We compare the accuracy of our enhanced LATTE layer algorithm selector with the normal LATTE selector and the heuristic API [10] provided by PyTorch for the 200 sub-models listed in the sub-model table on the three IID datasets and three non-IID datasets, and use the results from the on-device measurement API [12] of TensorFlow as ground truth (§ 3.2).

Results. In Figure 17, normal selector reaches 95.08%, 94.13%, and 95.84% accuracy on MobileNet, ResNet and Customized CNN, respectively. Our enhanced layer algorithm selector achieves 98.34%, 97.37%, and 98.18% accuracy respectively, which effectively augmented the efficiency and performance of the layer algorithm selection. Heuristic method only provides 73.24%, 73.03%, and 73.25% accuracy respectively, which is not sufficient for subsequent training time estimates.

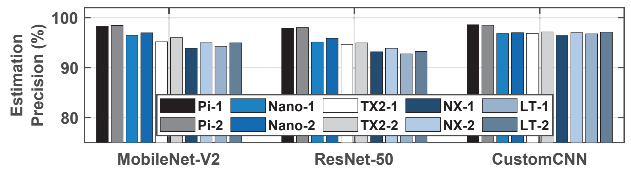


Fig. 18: Effectiveness of the training time estimator.

4.4.4 *Effectiveness of Training Time Estimator*

The training time estimator (§ 3.3) is also essential for the convergence speedup of LATTE. This experiment aims to show the applicability of our training time estimator across diverse devices.

Setups. We measure the estimator precision of the training latencies on ten different devices using the three IID datasets. It covers three device types, with two devices per type.

Results. In Figure 18, our training latency estimator leads to a precision of 94.26% to 98.41%, 92.73% to 97.99%, and 96.38% to 98.56% on MobileNet, ResNet and Customized CNN, respectively. The estimator precision of ResNet-50 on CIFAR-100 is slightly lower due to the more complex network architectures. In contrast, the precision of Customized CNN on HAR is higher, as there are fewer key layers. However, as shown in § 4.3, the precision of our training time estimator already significantly improves the overall training convergence.

4.5 Micro-benchmarks

4.5.1 *Impact of Deep Learning Frameworks*

This experiment validates that the layer algorithm is the key reason for variations in observed training times due to development-chain diversity, which can be resolved by LATTE. Specifically, we integrate LATTE with either TensorFlow or PyTorch (*i.e.*, replacing their default APIs) and measure the training latency of the same model on two identical devices, one developed with TensorFlow and the other with PyTorch. As shown in Figure 19, the wall-clock training times on the two devices using different deep learning frameworks are similar to each other. Recall that the wall-clock training time of a single epoch round with TensorFlow is only from 72.3% to 76% of that of PyTorch without LATTE calibration (see § 2.3).

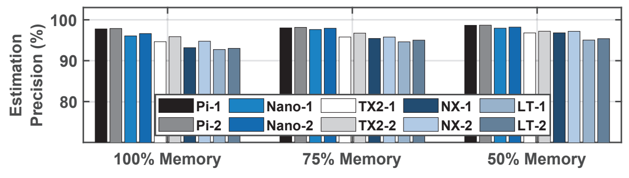


Fig. 20: Impact of the memory resource budget.

4.5.2 *Impact of Dynamic Memory*

This experiment tests LATTE under various runtime memory budgets. We employ memhog [6] to periodically change the available memory resources on each device, cycling from 100% available memory to 75% to 50%, and measure the precision of our training time estimator. Figure 20 shows that when the memory budget decreases, the estimation precision increases. This is because a tighter memory budget means a simpler sub-model should be selected. In Figure 22(a), the convergence time under dynamic memory is $2.18\times$ that of the normal state, and the model test accuracy decreases by 3.95%. This is because the drastic change in available memory force clients to conservatively select smaller sub-models, leading to a decrease in convergence speed and model test accuracy.

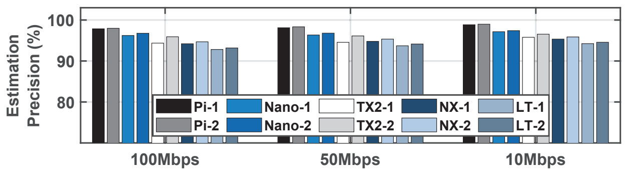


Fig. 21: Impact of the communication bandwidth.

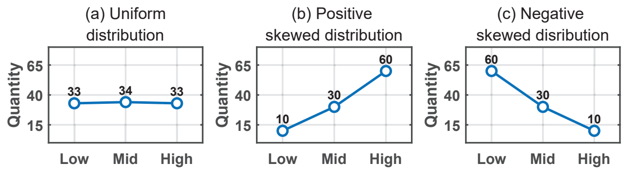


Fig. 23: Three distributions used in the simulation.

4.5.3 *Impact of Dynamic Bandwidth*

This experiment tests LATTE under different bandwidths. We assigned distinct bandwidth ranges to different devices (*e.g.*, Nano from 10 to 20Mbps, NX from 50 to 100Mbps, Laptop from 100 to 300Mbps) and periodically selected bandwidth values within these ranges. Additionally, we also introduced periodically changes to the bandwidth ranges to simulate realistic network conditions. We use Wondershaper [13] to change the communication bandwidth on each device. Figure 21 shows that the precision of training time estimator increases when the bandwidth decreases. This is because, given a fixed deadline, as the communication latency increases, the budget for model training is shorter, causing LATTE to choose simpler sub-models and thus estimate the training time more accurately. Figure 22(a) shows that under dynamic bandwidth conditions, both the convergence speed and model test accuracy are similar to the normal state. The convergence time increases slightly to $1.14\times$ of the stable state, and the model test accuracy decreases by 1.48%.

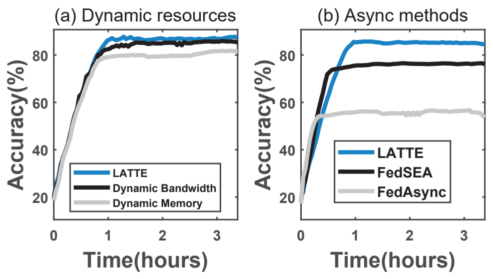


Fig. 22: Time-to-accuracy (a) under dynamic resources and (b) compared between different schemes.

4.5.4 *Comparison with Semi- and Fully-Asynchronous FL*

This experiment compares the performance of LATTE with FedAsync [74], a fully asynchronous FL method, and FedSEA [63], a state-of-the-art semi-asynchronous FL method. Figure 22(b) shows that both FedSEA and FedAsync converge faster than LATTE, since they do not wait for slow devices. However, their test accuracy is lower for aggregating slate model updates. For example, the test accuracy of FedSEA decreases by 9.19%, while the test accuracy of FedAsync suffers from a drastic drop by 28.96%.

In addition, it should be noticed that although LATTE is designed to enable synchronized training, its main designs also provide specific gains even in asynchronous FL scenarios. By enabling clients to select the fastest algorithm, it reduces local training times and mitigates update staleness, leading to better model accuracy. The server can also use LATTE to predict client completion times, allowing it to efficiently manage aggregation by ignoring updates that are likely to be too stale.

4.5.5 *Impact of Client Heterogeneity Distribution*

To understand this impact, we conduct a simulation-based evaluation. We simulate 100 clients using a server with 24 Xeon-Gold-6142 CPUs and 4 Tesla V100 32GB GPUs. All clients are initialized by Flower's VCE (Virtual Client Engine) function, half of which have TensorFlow installed and the other half have PyTorch installed. Clients are divided into three categories: low-end, mid-tier, and high-end, with a computational power ratio of 1:2:5 respectively. In this experiment, we train MobileNet on CIFAR-10 and perform the evaluation using three common device distributions, as shown in Figure 23.

Results. Figure 24(a) shows that under the uniform distribution, LATTE converges $2.07\times$ and $2.35\times$ faster than FedRolex and HetetoFL, respectively. Its test accuracy is also 2.3% and 9.36% higher than FedRolex and HetetoFL, respectively.

When the devices follow a positively skewed distribution, Figure 24(b) shows that all three methods can accelerate convergence and improve test accuracy. However, LATTE still outperforms the two baselines, with from $1.69\times$ to $2.15\times$ faster convergence and from 2.65% to 6.35% higher test accuracy. This is because more clients have sufficient computational power, allowing them to select larger and better sub-models, thus contributing more to the overall system performance.

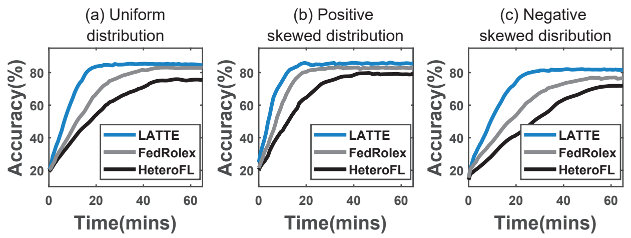


Fig. 24: Performance comparison under different device distributions in the simulation.

When the devices follow a negatively skewed distribution, LATTE still maintains its advantages. Compared with the two baselines, LATTE converges 1.89–2.23x faster and improves test accuracy by 5.09–10.1% in Figure 24(c). Under this distribution, more clients have insufficient computational power, forcing them to select smaller sub-models. However, LATTE's more accurate training time estimation enables it to select larger sub-models, leading to better performance. Overall, LATTE achieves faster convergence and higher model accuracy under different device heterogeneity distributions.

	ResNet-50	MobileNetV2	Customized CNN
Selector	191KB	191KB	191KB
Sub-model Table	509.6KB	197.6KB	52KB
Frequency Table	186KB	25KB	16KB
Selector	40ms	11ms	8ms

TABLE 4: LATTE memory and latency overhead on different models.

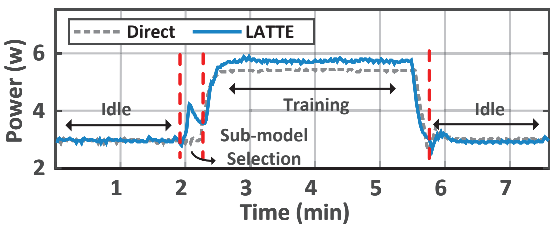


Fig. 25: Power consumption of LATTE.

4.6 System Overhead

Memory Overhead. LATTE introduces three types of memory/transmission overhead, including the parameters of selector, sub-model table, and frequency table, as shown in upper part of Table 4. In particular, each model used requires 191 KB memory of selector parameters. Sub-model tables take from 52 KB to 509.6 KB, on ResNet, MobileNet and Customized CNN, respectively. Frequency tables are optimized with delta encoding, only changes in training frequencies (delta values) are saved and transmitted, rather than the entire frequency table. Given that frequency changes are typically sparse and incremental, this reduces the transmitted data to a few hundred KB per round, which occupy from 16 KB to 186 KB on the aforementioned models in the same order. Such memory overhead is also considered within the memory budget during sub-model selection.

Latency Overhead. We have conducted experiments to measure the selector's runtime latency (from around 10ms to 40ms, as shown in lower part of Table 4.), confirming its negligible impact on overall training time (from 200s to 1000s). This is due to the selector is executed only once before the training begins, rather than repeatedly running in each iteration. Therefore, its latency does not accumulate as the number of training rounds increases. Results show that the selector's execution time is minimal (e.g., ¡0.1% of a training round's latency), ensuring LATTE remains lightweight.

Power Consumption. Low power consumption is important to mobile edge devices, and we measure the power consumption of LATTE on Jetson NX devices using the INA3221 power monitor [65]. As a baseline, we first measure the power consumption of direct training. We then measure LATTE for same duration of time. Figure 25 shows that the idle state of device consumes about 3 W. Before training, the power consumption of LATTE is slightly higher than the baseline, reaching 4.28 W, which is due to the fast sub-model allocation. During training, the power consumption is 5.84 W (by LATTE) and 5.6 W (by direct training), where LATTE itself consumes about 0.24 W only during resource polling.

5 Related Work

Heterogeneous Federated Learning. Federated learning with edge and IoT devices face the challenges of *data* and *system* heterogeneity [81]. Compared to centralized learning, data heterogeneity affects model accuracy [85], [50] and is often resolved by training personalized models [64], [39], [66], [55], [24], [83]. However, data heterogeneity is not the only factor affecting the model performance, and system heterogeneity also has a significant impact. System heterogeneity comes from the diverse computation and communication capabilities of devices, which can affect the training efficiency and is the focus of heterogeneous federated learning (HFL) [56].

HFL roughly falls into *model* and *system* level solutions. Model-level approaches [14], [25], [29], [58], [24], [54], [48], [61] assign sub-models tailored to the computation power of each device so that they can return the locally trained models almost simultaneously. The sub-models are extracted via masking [59], [25], [14], [24], [54], dropout [29], [58], knowledge distillation [86], [41], *etc.* Orthogonally, system-level strategies either perform client selection [37], [40], [43], [53] or adopt semi-asynchronous [63], [72] and fully asynchronous [33], [82], [20] model aggregation schemes to exclude or mitigate the impact of slow devices. Our work targets at model-level HFL solutions because client selection might ignore valuable data on slow devices whereas asynchronous model aggregation might affect model convergence. We focus on masking-based sub-model extraction for its widespread adoption and provide accurate local training time estimates so that the allocated sub-models match with the capabilities of devices. With the LATTE design, future methods to address data heterogeneity can be directly integrated to further improve FL performance. We provided a comparable Table 5 with four metrics to summarize the distinctions between our proposed solution and other related works, including the conference version of this paper. The ratios in the table derived from a qualitative comparison based both on the rationale of existing works and the empirical findings presented in our evaluation results. We can observe that our work holds a leading position across all metrics.

Related Works	Type	Methods	M1	M2	M3	M4
Oort [37]	System-lvl	Client Selection	◕	◑	◑	●
Papaya [33]		Asynchronous	●	●	◔	●
FedSEA [63]		Semi-Asyn	◑	◕	◑	◕
FedGen [86]	Model-lvl	KD	◔	●	◕	◔
Fjord [29]		Dropout	◑	●	◔	◔
HeteroFL [25]		Masking	◑	●	◑	◑
TailorFL [24]		Masking	◑	●	◕	◑
FedRolex [14]		Rolling Mask	◑	●	◕	◑
LATTE-Conf [68]		Rolling Mask	●	●	◕	◑
This Work	Model-lvl	Rolling Mask	●	●	◕	●

TABLE 5: Comparing our work with related HFL works in four metrics. M1: Overall training speed. M2: Clients coverage ratio, M3: Final model performance, M4: Dynamic device adaptivity

Model Latency Estimation. Evaluating the latency of model execution on specific devices is crucial for the optimization of model inference [21] and training [34]. Due to various model architectures, device types, and deep learning development chains, there is an increasing interest to predict the execution latency rather than measuring it exhaustively. These predictors model latency at the *network*, *layer*, or *operator* level [44]. For example, the number of FLOPs or MAC of the entire network is a common proxy for its latency [28], [47]. Mainstream predictors [17], [57], [35] resort to the layer level, where different layer features (*e.g.*, FLOPs or layer types) and hardware features are leveraged to train a regresssor to predict layer-wise latencies, which are then summed up as the overall model latency. Recent predictors [80] dive into the operator level to explicitly account for runtime optimizations such as operator fusion. However, our work lies in predicting training time, rather than **inference** or **compilation** as in prior works like [21] (focused on compile-time inference latency prediction) or [80] (focused on operator fusion during inference).

Our work is inspired by them yet aims at accurate latency prediction of model *training*. ElasticTrainer[32] proposed more fine-grained modeling of training time, but still overlooked the diversity of layer algorithm. By identifying layer algorithms as the previously overlooked feature, we devise a lightweight training latency estimator at the layer level. Our evaluations show such layer-level modeling matches with the current runtime optimization for model training on edge devices and delivers high accuracy despite its simplicity.

6 Discussion

We present the following discussion related to this paper.

1) Compatibility of LATTE in other model structures. In recent years, some more diverse model structures have emerged, including diffusion models and transformer models. We have investigated that newer model architectures typically have only a single algorithm implementation for their core components (e.g., attention mechanisms) within existing DL frameworks, which is very different from typical structure like CNNs. We hypothesize that this disparity exists because low-level, hardware-specific optimizations for these newer architectures are still in a early stage of exploration. As these models mature, we expect a similar evolution of specialized algorithms to occur. Our framework's model-agnostic design makes it inherently extensible and prepared to directly optimize these future models once multiple algorithm options become available.

2) Analysis between MLP and Bayes optimization. Our MLP-based selector is significantly more effective than Bayesian Optimization (BO) for on-the-fly layer algorithm selection for two key reasons. First, regarding latency, our MLP provides a near-instant decision via a single forward pass, having been trained offline; BO, conversely, would require a slow, iterative search for each layer at runtime, making it computationally prohibitive. Second, from a problem-framing perspective, our MLP learns a general, scalable mapping from configuration to algorithm. BO handles each layer as an isolated optimization, failing to learn a reusable model and restarting its expensive search for every new instance. Therefore, the MLP's ability to ”learn once and infer instantly” is essential for this application, whereas BO's ”search every time” methodology is fundamentally impractical in HFL scenario.

7 Conclusion

This paper presents LATTE, a new middleware design for accurate estimation of on-device training of deep learning models on mobile edge devices. Our core design insight is that even for the same model on the same device, training times can vary significantly due to runtime optimizations of deep learning frameworks. We solve this problem by designing a novel and more accurate layer algorithm selector with enhanced dataset and incorporating it into LATTE for accurate delay estimation. We further showcase the usability of our design in heterogeneous federated learning. Extensive experiments demonstrate significant performance improvements compared to state-of-the-art methods. A preliminary version of this study has been published in [68].

Acknowledgement

This work was sponsored by the GRF grant from Research Grants Council of Hong Kong under Grant CityU 11202623 and CityU 11206425. Zimu Zhou and Zhenjiang Li are the corresponding authors.

References

[1] Arm compute library. https://github.com/ARM-software/ComputeLibrary.
[2] bmon bandwidth monitor. https://github.com/tgraf/bmon.
[3] Cifar datasets. https://www.cs.toronto.edu/~kriz/cifar.html.
[4] CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/.
[5] cudnn. https://developer.nvidia.com/cudnn.
[6] memhog. https://github.com/afeinberg/memhog.
[7] Mkl-dnn. https://oneapi-src.github.io/oneDNN/v0/index.html.
[8] NVIDIA CUDA Compiler Driver NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.
[9] PyTorch. https://github.com/pytorch/pytorch.
[10] Pytorch runtime optimization. https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cudnn/Conv_v7.cpp.
[11] TensorFlow. https://github.com/TensorFlow/TensorFlow.
[12] Tensorflow runtime optimization. https://github.com/tensorflow/tensorflow/blob/v1.14.0/tensorflow/stream_executor/cuda/cuda_dnn.cc.
[13] Wondershaper. https://github.com/magnific0/wondershaper.
[14] Samiul Alam, Luyang Liu, Ming Yan, and Mi Zhang. Fedrolex: Model-heterogeneous federated learning with rolling sub-model extraction. In *Proc. of NeurIPS*, 2022.
[15] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In *Proc. of ESANN*, 2013.
[16] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Hei Li Kwing, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.
[17] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. In *Proc. of PMLR ACML*, 2017.
[18] Jiani Cao, Jiesong Chen, Chengdong Lin, Yang Liu, Kun Wang, and Zhenjiang Li. Practical gaze tracking on any surface with your phone. *IEEE Transactions on Mobile Computing*, 2024.
[19] Jiani Cao, Chengdong Lin, Yang Liu, and Zhenjiang Li. Gaze tracking on any surface with your phone. In *Proc. of ACM SenSys*, 2022.
[20] Ming Chen, Bingcheng Mao, and Tianyi Ma. Efficient and robust asynchronous federated learning with stragglers. In *Proc. of ICLR*, 2019.

[21] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proc. of USENIX OSDI*, 2018.

[22] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient gpu computing. In *2014 47th Annual IEEE/ACM international symposium on microarchitecture*, pages 343–355. IEEE, 2014.

[23] Allan de Barcelos Silva, Marcio Miguel Gomes, Cristiano André da Costa, Rodrigo da Rosa Righi, Jorge Luis Victoria Barbosa, Gustavo Pessin, Geert De Doncker, and Gustavo Federizzi. Intelligent personal assistants: A systematic literature review. *Elsevier Expert Systems with Applications*, 2020.

[24] Yongheng Deng, Weining Chen, Ju Ren, Feng Lyu, Yang Liu, Yunxin Liu, and Yaoxue Zhang. Tailorfl: Dual-personalized federated learning under system and data heterogeneity. In *Proc. of ACM SenSys*, 2022.

[25] Enmao Diao, Jie Ding, and Vahid Tarokh. Heterofl: Computation and communication efficient federated learning for heterogeneous clients. In *Proc. of ICLR*, 2020.

[26] In Gim and JeongGil Ko. Memory-efficient dnn training on mobile devices. In *Proc. of ACM MobiSys*, 2022.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*, 2016.

[28] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Prof. of Springer ECCV*, 2018.

[29] Samuel Horvath, Stefanos Laskaridis, Mario Almeida, Ilias Leontiadis, Stylianos Venieris, and Nicholas Lane. Fjord: Fair and accurate federated learning under heterogeneous targets with ordered dropout. In *Proc. of NeurIPS*, 2021.

[30] Ningning Hou, Yifeng Wang, Xianjin Xia, Shiming Yu, Yuanqing Zheng, and Tao Gu. Molora: Intelligent mobile antenna system for enhanced lora reception in urban environments. In *Proc. of ACM SenSys*, 2025.

[31] Kai Huang and Wei Gao. Real-time neural network inference on extremely weak devices: agile offloading with explainable ai. In *Proc. of ACM MobiCom*, 2022.

[32] Kai Huang, Boyuan Yang, and Wei Gao. Elastictrainer: Speeding up on-device training with runtime elastic tensor selection. In *Proc. of ACM MobiSys*, 2023.

[33] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, Kaikai Wang, Anthony Shoumikhin, Jesik Min, and Mani Malek. Papaya: Practical, private, and scalable federated learning. In *Proc. of MLSys*, 2022.

[34] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proc. of MLSys*, 2019.

[35] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the computational cost of deep learning models. In *Proc. of IEEE BigData*, 2018.

[36] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *Springer International Journal of Computer Vision*, 2020.

[37] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *Proc. of USENIX OSDI*, 2021.

[38] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: an efficient federated learning framework for heterogeneous mobile clients. In *Proc. of ACM MobiCom*, 2021.

[39] Ang Li, Jingwei Sun, Xiao Zeng, Mi Zhang, Hai Li, and Yiran Chen. Fedmask: Joint computation and communication-efficient personalized federated learning via heterogeneous masking. In *Proc. of ACM SenSys*, 2021.

[40] Chenning Li, Xiao Zeng, Mi Zhang, and Zhichao Cao. Pyramidfl: A fine-grained client selection framework for efficient federated learning. In *Proc. of ACM MobiCom*, 2022.

[41] Daliang Li and Junpu Wang. Fedmd: Heterogenous federated learning via model distillation. *arXiv preprint arXiv:1910.03581*, 2019.

[42] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 2020.

[43] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *Proc. of MLSys*, 2020.

[44] Ying Li, Yifan Sun, and Adwait Jog. Path forward beyond simulators: Fast and accurate gpu execution time prediction for dnn workloads. In *Proc. of IEEE/ACM MICRO*, 2023.

[45] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. A workload-aware dvfs robust to concurrent tasks for mobile devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023.

[46] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proc. of IEEE ICCV*, 2017.

[47] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *Proc. of ICLR*, 2019.

[48] Ruixuan Liu, Fangzhao Wu, Chuhan Wu, Yanlin Wang, Lingjuan Lyu, Hong Chen, and Xing Xie. No one left behind: Inclusive federated learning over heterogeneous devices. In *Proc. of KDD*, 2022.

[49] Sicong Liu, Bin Guo, Cheng Fang, Ziqi Wang, Shiyan Luo, Zimu Zhou, and Zhiwen Yu. Enabling resource-efficient aiot system with cross-level optimization: A survey. *IEEE Communications Surveys & Tutorials*, 2023.

[50] Zili Lu, Heng Pan, Yueyue Dai, Xueming Si, and Yan Zhang. Federated learning with non-iid data: A survey. *IEEE Internet of Things Journal*, 2024.

[51] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges. *IEEE Internet of Things Journal*, 2022.

[52] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proc. of PMLR AISTATS*, 2017.

[53] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *Prof. of IEEE ICC*, 2019.

[54] Chaoyue Niu, Fan Wu, Shaojie Tang, Lifeng Hua, Rongfei Jia, Chengfei Lv, Zhihua Wu, and Guihai Chen. Billion-scale federated learning on mobile clients: A submodel design with tunable privacy. In *Proc. of ACM MobiCom*, 2020.

[55] Xiaomin Ouyang, Zhiyuan Xie, Jiayu Zhou, Jianwei Huang, and Guoliang Xing. Clusterfl: a similarity-aware federated learning system for human activity recognition. In *Proc. of ACM MobiSys*, 2021.

[56] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. Federated learning for computationally-constrained heterogeneous devices: A survey. *ACM Computing Surveys*, 2023.

[57] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *Proc. of ICLR*, 2016.

[58] Xinchi Qiu, Javier Fernandez-Marques, Pedro PB Gusmao, Yan Gao, Titouan Parcollet, and Nicholas Donald Lane. Zerofl: Efficient on-device training for federated learning with local sparsity. In *Proc. of ICLR*, 2021.

[59] Martin Rapp, Ramin Khalili, Kilian Pfeiffer, and Jörg Henkel. Distreal: Distributed resource-aware learning in heterogeneous systems. In *Proc. of AAAI*, 2022.

[60] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. of IEEE CVPR*, 2018.

[61] Leming Shen, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. Fedconv: A learning-on-model paradigm for heterogeneous federated clients. In *Proc. of ACM MobiSys*, 2024.

[62] Leming Shen, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. Hierarchical and heterogeneous federated learning via a learning-on-model paradigm. *IEEE Transactions on Mobile Computing*, 2025.

[63] Jingwei Sun, Ang Li, Lin Duan, Samiul Alam, Xuliang Deng, Xin Guo, Haiming Wang, Maria Gorlatova, Mi Zhang, Hai Li, and Yiran Chen. Fedsea: A semi-asynchronous federated learning framework for extremely heterogeneous devices. In *Proc. of ACM SenSys*, 2022.

[64] Alysa Ziying Tan, Han Yu, Lizhen Cui, and Qiang Yang. Towards personalized federated learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
[65] Texas Instruments Inc. INA3221 power monitor. https://www.ti.com/product/INA3221, 2016.
[66] Linlin Tu, Xiaomin Ouyang, Jiayu Zhou, Yuze He, and Guoliang Xing. Feddl: Federated learning via dynamic layer sharing for human activity recognition. In *Proc. of ACM SenSys*, 2021.
[67] Kun Wang, Jiani Cao, Zimu Zhou, and Zhenjiang Li. Swapnet: Efficient swapping for dnn inference on edge ai devices beyond the memory budget. *IEEE Transactions on Mobile Computing*, 2024.
[68] Kun Wang, Zimu Zhou, and Zhenjiang Li. Latte: Layer algorithm-aware training time estimation for heterogeneous federated learning. In *Proc. of ACM MobiCom*, 2024.
[69] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proc. of ACM MobiCom*, 2021.
[70] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proc. of ACM MobiSys*, 2022.
[71] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.
[72] Wentai Wu, Ligang He, Weiwei Lin, Rui Mao, Carsten Maple, and Stephen Jarvis. Safa: A semi-asynchronous protocol for fast federated learning with low overhead. *IEEE Transactions on Computers*, 2020.
[73] Yu Xianjia, Jorge Peña Queralta, Jukka Heikkonen, and Tomi Westerlund. Federated learning in robotic and autonomous systems. *Elsevier Procedia Computer Science*, 2021.
[74] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934*, 2019.
[75] Huatao Xu, Pengfei Zhou, Rui Tan, Mo Li, and Guobin Shen. Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications. In *Proc. of ACM SenSys*, 2021.
[76] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology*, 2019.
[77] Shiming Yu, Xianjin Xia, Ningning Hou, Yuanqing Zheng, and Tao Gu. Revolutionizing lora gateway with xgate: Scalable concurrent transmission across massive logical channels. In *Proc. of ACM MobiCom*, 2024.
[78] Shiming Yu, Xianjin Xia, Ningning Hou, Yuanqing Zheng, and Tao Gu. Xgate: Scaling lora communications to massive logical channels. *IEEE Transactions on Networking*, 2025.
[79] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2015.
[80] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proc. of ACM MobiSys*, 2021.
[81] Tuo Zhang, Lei Gao, Chaoyang He, Mi Zhang, Bhaskar Krishnamachari, and A Salman Avestimehr. Federated learning for the internet of things: Applications, challenges, and opportunities. *IEEE Internet of Things Magazine*, 2022.
[82] Tuo Zhang, Lei Gao, Sunwoo Lee, Mi Zhang, and Salman Avestimehr. Timelyfl: Heterogeneity-aware asynchronous federated learning with adaptive partial training. In *Proc. of IEEE CVPR*, 2023.
[83] Wenhao Zhang, Zimu Zhou, Yansheng Wang, and Yongxin Tong. Dm-pfl: Hitchhiking generic federated learning for efficient shift-robust personalization. In *Proc. of KDD*, 2023.
[84] Hanhan Zhou, Tian Lan, Guru Prasadh Venkataramani, and Wenbo Ding. Every parameter matters: Ensuring the convergence of federated learning with dynamic heterogeneous models reduction. In *Proc. of NeurIPS*, 2024.
[85] Hangyu Zhu, Jinjin Xu, Shiqing Liu, and Yaochu Jin. Federated learning on non-iid data: A survey. *Elsevier Neurocomputing*, 2021.
[86] Zhuangdi Zhu, Junyuan Hong, and Jiayu Zhou. Data-free knowledge distillation for heterogeneous federated learning. In *Prof. of ACM ICML*, 2021.

Kun Wang received the B.E. degree from Xidian University, Xi'an, China, in 2020, and the Ph.D. degree from the City University of Hong Kong, Hong Kong, China, in 2024. He is currently a Postdoctoral Fellow at the Department of Computer Science, City University of Hong Kong. His research interests lie in designing and building efficient on-device AI systems, including on-device inference, on-device training and on-device large language models.

Zimu Zhou received the B.E. from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2011, and the Ph.D. from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, in 2015. He is currently an Assistant Professor at the Department of Data Science, City University of Hong Kong. His research focuses on mobile and ubiquitous computing.

Zhenjiang Li received the B.E. degree from Xi'an Jiaotong University, China, in 2007, and the M.Phil. and Ph.D. degrees from the Hong Kong University of Science and Technology, Hong Kong, China, in 2009 and 2012, respectively. He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong. His research interests include edge/embedded AI systems, Artificial Internet of Things (AIoT), and low-power systems.