Collate: Collaborative Neural Network Learning for Latency-Critical Edge Systems

Shuo Huai[‡§], Di Liu[§], Hao Kong[‡§], Xiangzhong Luo[‡], Weichen Liu[‡],
Ravi Subramaniam[†], Christian Makaya[†] and Qian Lin[†]
[‡]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[§]HP-NTU Digital Manufacturing Corporate Lab, Nanyang Technological University, Singapore
[†]HP Inc., Palo Alto, California, USA
Email: {shuo001, liu.di, kong.hao, xiangzho001, liu}@ntu.edu.sg, {ravi.subramaniam, christian.makaya, qian.lin}@hp.com

***Abstract*—Federated Learning (FL) empowers multiple clients to collaboratively learn a model, enlarging the training data of each client for high accuracy while protecting data privacy. However, when deploying FL in real-time edge systems, the heterogeneity of devices among systems has a severe impact on the performance of the inferred model. Existing optimizations on FL focus on improving the training efficiency but fail to speed up inference, especially when there is a latency constraint. In this work, we propose *Collate*, a novel training framework that collaboratively learns heterogeneous models to meet the latency constraints of multiple edge systems simultaneously. We design a dynamic zeroizing-recovering method to adjust each local model architecture for high accuracy under its latency constraint. A proto-corrected federated aggregation scheme is also introduced to aggregate all heterogeneous local models, satisfying the latency constraint of different systems with only one training process and maintaining high accuracy. Extensive experiments indicate that, compared to state-of-the-art methods and under a latency constraint, our extended models can improve the accuracy by 1.96% on average, and our shrunk models can also obtain a 3.09% accuracy improvement on average, with almost no extra training overhead. The related codes and data will be available at https://github.com/ntuliuteam/Collate.**

***Index Terms*—edge devices, edge intelligence, neural network learning, federated learning, inference efficiency**

I. Introduction

Deep Neural Networks (DNNs) have brought significant breakthroughs in many different applications, such as image recognition and natural language processing [1]. With the emphasis on data privacy and concerns over transmission stability, current DNN applications are increasingly deployed on edge devices, such as autonomous vehicles, healthcare devices, etc [1]. Meanwhile, DNN models require a huge amount of training data to improve accuracy [2], but data in most industries are protected by privacy laws and thus are required to be in the form of isolated islands.

Federated Learning (FL) is designed to coordinate multiple clients to train a DNN model collaboratively without sharing their original local data, and it is capable of preserving the data privacy and achieving better accuracy than each individual client training with only its local data [3]. This addresses the issue of insufficient individual training data. However, in addition to accuracy, latency is also an important metric for edge intelligent systems [4]. With diverse edge devices that demonstrate different computational capabilities emerging, the identical model trained by FL is not efficient for all participating edge systems and cannot achieve a good balance between latency and accuracy. As demonstrated in Fig. 1, when deploying the model into various edge systems, the latency differs from 13.6 ms to 236.0 ms. When this application has a latency constraint (e.g., 30.0 ms), some systems can infer more complex models for higher accuracy, whereas others cannot even deploy the model due to the limited memory and computational resources.

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from the industry partner, HP Inc., through the HP-NTU Digital Manufacturing Corporate Lab (I1801E0028).

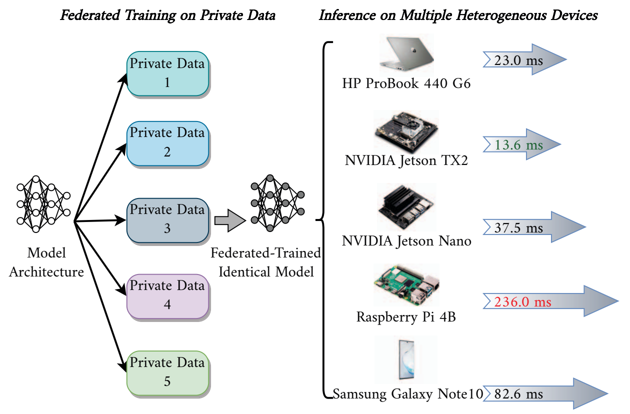

Fig. 1. Latency of the federated-trained AlexNet on various edge systems.

Take a practical scenario from WeBank as an example [5]. They need an edge FL framework in which surveillance video data collected and stored in the edge cloud of each surveillance company are not required to be uploaded to a central cloud for centralized model training. After each local training iteration, only the model parameters from each surveillance company are sent to the FL server for aggregation. The final federated-trained model is distributed to the participating surveillance companies for object detection. However, different companies can use distinct edge devices and this application features real-time, so it is necessary to train heterogeneous models for different companies to meet their latency constraints.

Some efforts have been made to use different DNN architectures to fit various clients in the training stage, known as Heterogeneous FL [6]–[10]. These methods mainly fall into two categories: one is to fine-tune models for different clients from an identical global model, and the other is to directly learn from heterogeneous models without the same global model. However, these existing approaches are designed for accelerating the training stage and cannot optimize the inference latency directly. When the first one is employed in a latency-critical system, it can only train a specific network for one system at a time until it trains for all systems using multiple FL processes, which imposes enormous training overhead. Although the second one can directly provide heterogeneous model architectures for different systems, it requires some public datasets for transfer learning [9] or an extra dataset for prototype learning [10]. Moreover, the absence of the same global model lead to an accuracy drop of up to 10% [11]. To guarantee the generality and accuracy of the FL scheme, our method should be based on the first one.

Meanwhile, when optimizing models under the latency constraint, we should not only reduce latency for low-end systems but also extend models for powerful systems (e.g., Jetson TX2 in Fig. 1) to improve their accuracy [12], in contrast to existing heterogeneous FL that only reduces the training cost by shrinking models. Thus, we integrate the model extension into our learning framework to better utilize each client. To our best knowledge, this is the first paper to optimize FL to simultaneously meet the latency constraints of all participating systems while obtaining high accuracy. Specifically, our main contributions are summarized as:

- We propose a novel model learning framework, *Collate*, that cultivates optimal DNN architectures collaboratively for multiple edge systems to obtain higher accuracy and satisfy their latency constraints with only one training process.
- We present a proto-corrected aggregation scheme in the global training process to effectively aggregate all heterogeneous models from each edge system for higher accuracy.
- We design a latency-aware local training scheme by a dynamic zeroizing-recovering training process. It extends the exploration space of *Collate* to discover the optimal DNN architecture for each edge system.
- We demonstrate the effectiveness of *Collate* with extensive experiments. Compared to the state-of-the-art methods and under the same latency constraints, our extended models can improve the accuracy by 1.96% on average, and the accuracy of shrunk models outperforms others by 3.09% on average.

II. Background & Related Work

In this section, we introduce the preliminaries of FL, heterogeneous FL and latency prediction, including the advantages and disadvantages of some related work.

A. Federated Learning

In the traditional FL algorithm (FedAvg [13]), multiple edge systems collaborate to train the same model on their respective local training data to solve issues of data isolated islands and protect privacy. During this training step, different systems transmit model parameters to the server after every le local training epochs. Then the server executes model aggregation. Considering this FL process includes n systems, the loss function for system i is f_i. Let w represent the model parameter, and then the training optimization goal becomes:

$$\min_{w} f(w) := \frac{1}{n}\sum_{i=1}^{n} f_i(w) \tag{1}$$

And the training process can be formulated as:

$$w_i^{t+(e+1)} = w_i^{t+e} - \eta_i^{t+e}\nabla f_i(w_i^{t+e}) \tag{2}$$

$$w^{t+1} = \frac{1}{n}\sum_{i=1}^{n} w_i^{t+le} \tag{3}$$

where w_i^{t+e} denotes the local model parameter on system i after t communication rounds and e local training steps. η is the learning rate, and ∇ represents the derivative. From Eq. (3), we can see that the traditional FL [13] is based on the fact that all local model architectures of different systems are the same for aggregation.

B. Heterogeneous Federated Learning

Two major challenges involved in the traditional FL process are statistical heterogeneity and hardware heterogeneity among different systems [14]. Statistical heterogeneity means the data on different systems are not independent and identically distributed (Non-IID) [15], and different systems have different information to learn, while hardware heterogeneity refers to various devices participating in the FL process having different computational abilities. Different heterogeneous FL methods are proposed for these two challenges.

Statistical Heterogeneity: Most existing works on statistical heterogeneity aim to achieve higher accuracy on each local data. They use different or identical model architectures with different parameters for each participating system by customizing the global model. Wang *et al.* [16] proposed to fine-tune some or all parameters of a trained global model using each client's local data. Jiang *et al.* [17] proposed to combine meta-learning with FL for customizing models to recognize each local data pattern. Khodak *et al.* [18] proposed to improve the accuracy of statistically heterogeneous FL via online convex optimization theory and meta-learning. These methods focus on improving the accuracy of local models on the local dataset of each system. They are applied to the scenario where most of the inference data of each system match the pattern of its corresponding training set. This paper aims to learn different local models for all participating edge systems to meet their latency requirements. Thus, when considering statistical heterogeneity, these aforementioned methods can be used as the back-end of our method to achieve higher accuracy.

Hardware Heterogeneity: Current research on hardware heterogeneity concentrates on simplifying the model architecture to accelerate training and reduce the model size on devices with low computational ability. Caldas *et al.* [19] proposed to randomly select small subsets of the global model with the expected model volume to provide a reduction in both client-to-server communication and local computation during the training. Li *et al.* [20] proposed Hermes to find small sub-networks for some devices by applying the structured pruning. Jeong *et al.* [21] proposed to use knowledge distillation to reduce model size in some low-end devices. Xu *et al.* [7] proposed Helios to accelerate the devices with weak computational capacities by dynamically compressing the global model into an expected volume during the training stage. Diao *et al.* [8] proposed HeteroFL to assign models with different computational complexities to each device in advance according to the computational ability of each device. Also, they proposed a method to aggregate heterogeneous local models to produce a global model during the training stage.

These methods are proposed to reduce the training time or the communication cost when applying on-device training. However, FL is not just on-device training [22]. And typically, the usage (inference) of a model lasts longer than its generation (training). Thus, inference optimization is more critical for the FL framework. Although some of these methods can be modified to accelerate inference on low-end devices, these methods designed for the global model lead to a large accuracy loss for each local model. Meanwhile, these methods did not consider improving accuracy for powerful devices by extending models, and their aggregation methods introduce an accuracy drop to large models, as illustrated in Section IV.

C. Latency Prediction

Optimizing models based on latency can better explore hardware characteristics, providing additional advantages in the trade-off between accuracy and latency [23]. To obtain high accuracy under latency constraints, we dynamically adjust the local model architectures

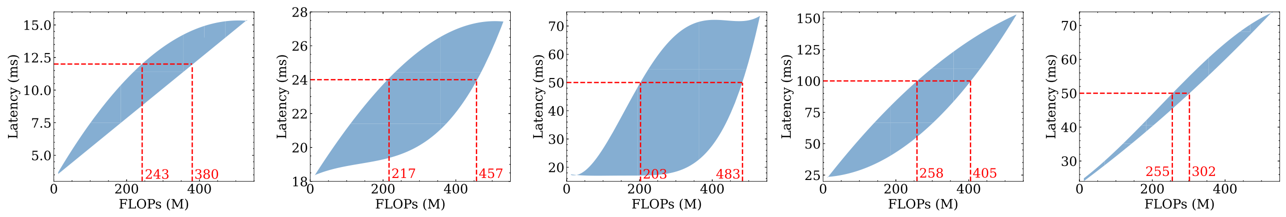

Fig. 2. Relationship between latency and FLOPs of ResNet variants on devices (From left to right are HP Probook, Jetson TX2, Jetson Nano, Pi and Note10).

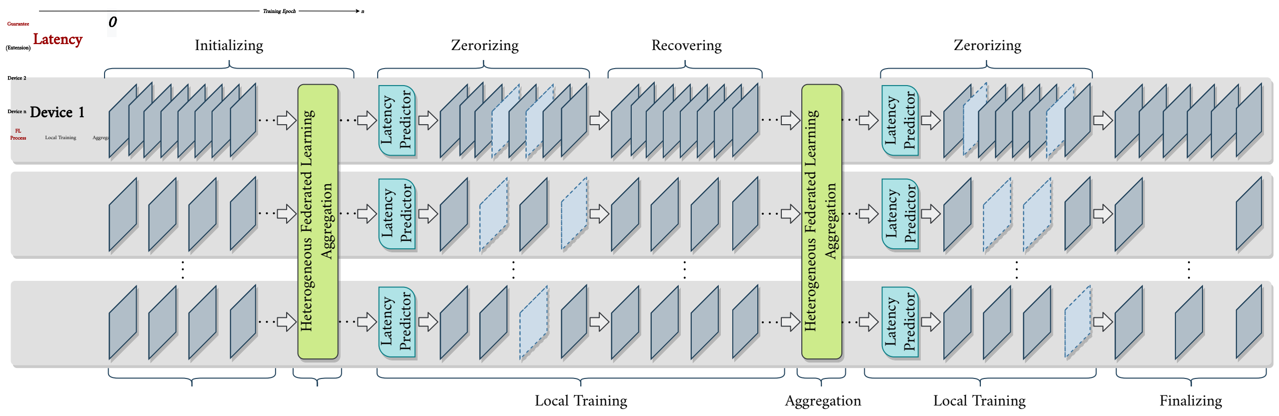

Fig. 3. The process of *Collate*. The top depicts the latency guarantee component, while the bottom shows the heterogeneous FL component.

during their local training, and the new model architecture needs to satisfy the latency constraint. However, measuring latency on a device will interrupt the training process and usually takes minutes [24], especially for off-device training [22]. And as shown in Fig. 2, the relationship between latency and floating-point operations (FLOPs) of a model is weak, making it unlikely to calculate the latency with an arithmetic function with respect to the FLOPs. A latency predictor is necessary for latency-critical systems.

As the overhead of on-device measurement is huge, previous works for latency-critical systems use hardware simulator-based [25], look-up table (LUT)-based [23], [24], [26] or neural network-based [27], [28] predictors to estimate the latency of DNN on a certain device. Most commercial devices are black boxes to users, so it is difficult to emulate hardware by analyzing its resources and scheduling algorithms for building hardware simulators. On the other hand, LUT-based predictors calculate the latency of DNN by summing up the recorded latency of each layer, resulting in that it only suiting models made up of pre-defined layer structures. During our latency-critical learning process, the number of input channels and kernels, and other size-related parameters of each layer will be updated continuously to find the best model architecture under the latency constraint. It is impractical for LUT to preserve the whole design space. Thus, in this work, we follow [28] to build a hardware-customized latency predictor based on the Backpropagation (BP) [29] neural network.

III. Methodology

In this section, we demonstrate our model learning framework, which collaboratively trains heterogeneous DNNs for different edge devices to meet their respective latency constraints simultaneously, while optimizing the model architecture for high accuracy. As shown in Fig. 3, this framework mainly includes two parts: the latency guarantee for each participant and the heterogeneous federated learning process. We also employ model extension for powerful clients, and we use the latency predictor to optimize this flow. Fig. 3 also illustrates the change of local models on each device during our training process, where we use kernels in a layer as an example to show this training process, and each oblique rectangle represents a kernel in this layer. The dashed line means this kernel is shielded in the current epoch. In the following, we first briefly introduce the hardware-customized latency predictor and the model extension. Then we present the latency guarantee, which is integrated into the local training of the FL process with almost no extra training overhead. Finally, we introduce our heterogeneous FL, mainly focusing on the heterogeneous aggregation algorithm.

A. Latency Predictor & Model Extension

Latency Predictor: Based on [28], we build a hardware-customized latency predictor using a three-layer BP network. The BP network is a lightweight neural network, and its forward propagation is shown in Eq. (4), where x is the vector of inputs, W^L is the weight matrix, L is the index of layers, and σ^L is the activation function at layer L. Thus, hundreds of multiplication and addition operations are enough to obtain the latency with the BP network. Also, the updating scheme of the BP network is simple, so its training is not time-consuming.

$$g(x) = \sigma^2(W^2\sigma^1(W^1\sigma^0(W^0x))) \tag{4}$$

We first build a pool of models, which comprises thousands of single-layer models with various operations like Convolution and Pooling, configured with different configurations, to train the latency predictor. The latency of single-layer models is only a few milliseconds, so their measurement does not take much time. After training, by feeding the configuration of a layer, the latency predictor can predict its latency. For a real model, adding the latency of all

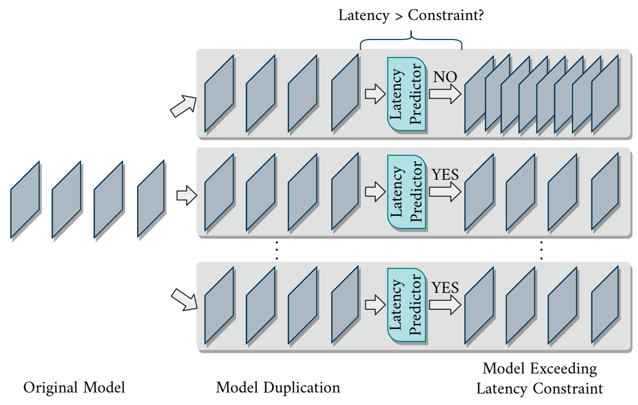

Fig. 4. The model Extension scheme of *Collate*.

layers in the latency-weighted longest path can get the whole model latency. The latency predictor is trained in advance for each device and is used in model learning. Assessing the latency predictor of Jetson TX2 with real models shows its variation is about 6.12%.

Model Extension: In this work, we directly optimize models under the inference latency constraint, so *Collate* not only reduces latency for low-end systems but also extends models for powerful systems to improve accuracy. As illustrated in Fig. 4, when the latency of the original model is lower than the latency constraint, we can uniformly extend the model [30] on this device to obtain a larger model. The uniform extension means all layers are extended with a constant width multiplier, which enables the new model to exceed the system latency constraint. Then our following learning method can ensure the latency constraint is satisfied again. The model extension improves the accuracy for powerful devices (see Section IV), but it introduces heterogeneity among different local models.

B. Latency Constraint Guarantee – Local Training

As shown in Fig 3, the latency guarantee flow is integrated into the local training of the FL process. This flow can be divided into three phases: Initializing, Zeroizing, and Recovering. Algorithm 1 demonstrates the details of local training with the latency constraint guarantee.

We first train the model for some communication rounds, called Initializing process, to let model parameters have informative values to reflect their contributions (*Line 2-4*) instead of randomly initialized numbers. The proto-training will be introduced in the next section. We design a trainable mask layer following each convolution layer, and each value δ in the mask layer is multiplied into the output of each kernel in its previous convolution layer. Thus, the contribution metric we define is that kernels with larger absolute mask values provide more impact on the final accuracy. And zeroizing the mask value means removing the corresponding kernel from the previous convolution layer. Note that mask values can be fused into the weights of kernels before inference to eliminate their overhead. In the Zeroizing process, we first sort all contribution factors δ according to their absolute values to get the global contribution rank of kernels in the entire local model (*Line 8*). During the whole training process, these contribution factors are jointly optimized with the network weights, hence the training scheme can automatically identify the contribution of each kernel. Next, we calculate the contribution factor threshold (*Line 10*) by the latency predictor (*Line 9*) and the

Algorithm 1: Latency Guarantee – LocalTrain

```
Input: model, training settings, current_round: $c$, start_zero
       round: $s$, local_epoch: $le$, latency constraint: $l$,
       prototype array: $pt$.
Output: updated local model, output prototype
 1: Load private training data;
 2: if $c < s$ then
 3:     $train\_with\_proto(pt)$ for $le$ epochs;
 4: else
 5:     for $e \leftarrow 1$ to $le$ do
 6:         $train\_with\_proto(pt)$;
 7:         if $e\%2 \, != 0$ or $e$ is the last epoch then
 8:             $Con\_rank \leftarrow$ Sort kernels by $|\delta|$;
 9:             $Shrink\_ratio \leftarrow$ Predictor($l, Con\_rank$);
10:             Find threshold of $\delta$ by $Con\_rank$, $Shrink\_ratio$;
11:             Zeroize $\delta_i$ if $\delta_i <$ threshold;
12:         end if
13:     end for
14: end if
```

contribution rank of the entire model. Finally, we zeroize those δ_i corresponding to the unimportant parts with these thresholds (*Line 11*) to temporarily remove these weights for the current epoch. After this Zeroizing process, we extract a compact model that meets the latency constraint.

Following each Zeroizing process, we propose a Recovering process (*Line 7*) to allow those zeroized parameters (i.e., δ_i) to recover instead of being permanently removed. If any weights that are eliminated in the previous Zeroizing process but are potentially essential, the Recovering process can help them escape from zero and play a crucial part in the subsequent training process, the aggregation process, and even the inference processes. The Recovering process gives our method a chance to learn more efficient architecture with higher accuracy. Eq. (5) indicates the calculation formula in a convolution layer with δ and activation function, where L is the index of layers, A^L is the output of layer L, W^L is the weight between layer $L-1$ and L, and σ^L is the activation function at layer L. $\circledast$ denotes the convolution operation.

$$A^L = \sigma^L(\delta^L \cdot (A^{L-1} \circledast W^L)) \tag{5}$$

In the Recovering processing, we mainly focus on the updating of δ^L that are zeroized during the Zeroizing process. Eq. (6) shows the gradients of δ^L to the final loss function.

$$\frac{\partial loss}{\partial \delta^L} = \frac{\partial loss}{\partial A^L} \cdot \sigma^{L'} \cdot (A^{L-1} \circledast W^L) \tag{6}$$

Since we have zeroized some values in δ^L and the most often used activation function is *ReLU*, the corresponding output of σ^L is likewise zero under *ReLU*. Although *ReLU* is not differentiable at zero, it is widely accepted that its derivative is also zero. Thus, we can easily derive Eq. (7).

$$\delta^L = 0 \rightarrow \sigma^{L'} = 0 \rightarrow \frac{\partial loss}{\partial \delta^L} = 0 \tag{7}$$

However, most training algorithms employ momentum in the optimizer to speed up convergence, as the momentum accumulates the gradients of the past steps to determine the direction to go in, rather than using only the gradient of the current step [31]. Eq. (8)-(9) illustrates the updating rules of weight with momentum, in which η is the learning rate, z_k is the updated value of the last step, and m

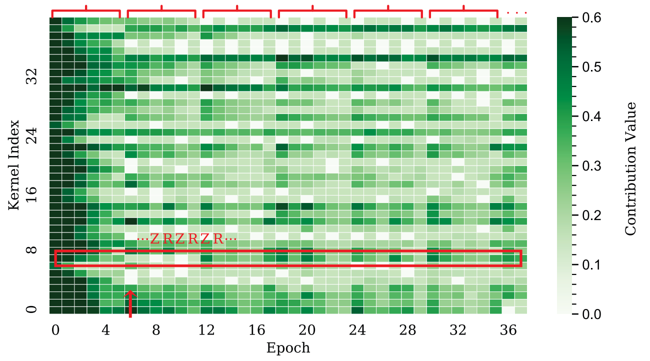

Fig. 5. The contribution values of a local model. ↑ points to the communication round that starts zeroizing. ⌒ packages each local training.

is the accumulation coefficient. Thus, w_k are easily updated to w_{k+1} by combining current and past gradients.

$$z_{k+1} = m \cdot z_k + \frac{\partial loss}{\partial w_k} \tag{8}$$

$$w_{k+1} = w_k - \eta \cdot z_{k+1} \tag{9}$$

Therefore, even though the current gradients of δ^L are zero, they can recover from zero by following the last updating directions. This process is fully automated and efficient, as no additional steps are required. These kernels zeroized in the previous zero epoch may become significant and we will not zeroize them in the future Zeroizing process. Thus, the architecture of the local model is changed, as illustrated in Fig. 5. Moreover, the training process tries to minimize loss and improve accuracy, allowing the model architecture to improve for better accuracy. Our local learning gradually cultivates an optimal architecture for each system through the training process to maximize accuracy under the latency constraint.

Training Overhead: Compared to the local training of traditional FL, we add mask layers and a zeroizing process. As they are both related to the kernels in convolution layers, we mainly focus on the convolution. The time complexity of convolution layers for an input is $O(\sum_{l=1}^{L} M_l^2 K_l^2 C_{l-1} C_l)$, where L is the number of convolution layers, C_l is the number of the kernels in layer l, that is, the number of output channels of this layer, M_l is the output size of layer l and K_l represents the kernel size of layer l. Mask layers have a time complexity of $O(\sum_{l=1}^{L} M_l^2 C_l)$, which is much lower than convolution operations. The zeroizing operation includes a sorting step with a time complexity of $O(nlogn), n = \sum_{l=1}^{L} C_l$. Generally, $logn$ is much smaller than $M_l^2 K_l^2 C_{l-1}$ of each layer l, so its complexity is lower than the convolution layers. The latency predictor only includes hundreds of multiplication and addition operations, which is much smaller than convolution operations. Thus, the overhead of our local training can be neglected.

C. Heterogeneous Aggregation – Global Training

FL maintains a global model from all locally trained models, combining all patterns learned from each private dataset. The parameters of local models are represented by $\{\ W_l^1, W_l^2, ..., W_l^n\ \}$, where l means different layers, and n is the total number of local models. The global model receives all local model parameters and aggregates them into a global model W_{gl}. As shown in Fig. 3, after several local training epochs, an aggregation process is conducted to combine all local parameters. Each local training - aggregation

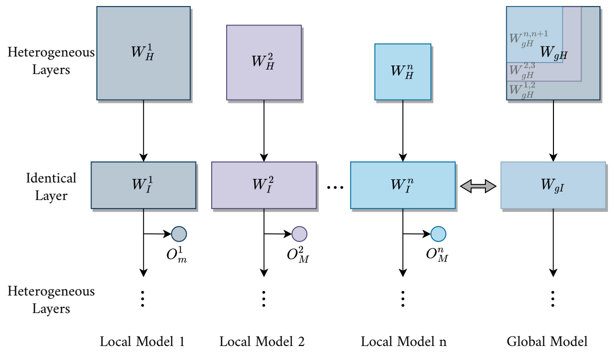

Fig. 6. Heterogeneous federated learning aggregation scheme of *Collate*.

is a communication round, and it iterates multiple rounds. In the traditional identical FL algorithm, the aggregation is formulated as $W_{gl}^{t+1} = \frac{1}{n}\sum_{i=1}^{n}(W_l^i)^{t+le}$ at the end of iteration t. At iteration $t+1$, W_{gl}^{t+1} is transmitted to each local model and sets the local model parameters as $(W_l^i)^{t+1} = W_{gl}^{t+1}$.

In this work, our primary motivation is to learn models for each system to meet its latency constraint while obtaining as much accuracy as possible. The details of our global training are shown in Algorithm 2. We exploit model extension (*Line 4*) and shrinking (*Line 10*) to adjust each local model to suit its corresponding system, and these local models have a similar model architecture. Parameters of the smaller model are a subset of parameters of the larger model. Inspired by HeteroFL [8], we first use an intuitive way to aggregate these heterogeneous local models (*Line 13-16*), as shown in the rectangle in Fig. 6. The model architecture mainly includes two parts, where the heterogeneous layers are shrunk or extended to change the latency, and the identical layer is for proto-correction. The identical layer is a fully-connected layer, and this layer has the same shape among all local models. It can be aggregated following the traditional FL aggregation scheme as all local models share the same architecture. We focus more on the aggregation of parameters W_H in heterogeneous layers. For notational convenience, we drop the iteration index t. Also, we aggregate parameters layer by layer, and we take one heterogeneous layer as an example. W_H^i is the parameters of a heterogeneous layer of local model i, and $W_H^i \setminus W_H^j$ means the parameters in W_H^i but not in W_H^j, $W_H^i \cap W_H^j$ represents the parameters in both W_H^i and W_H^j, while $W_H^i \cup W_H^j$ represents the parameters in W_H^i or W_H^j. n is the total number of local models, and n_a is the number of local models that come before local model a in the complexity rank (*Line 6*). Thus, the aggregation process can be represented in Eq. (10)-(11).

$$\begin{aligned} W_{gH}^{a,a+1} &= \frac{1}{n_{a+1}} \sum_{i=1}^{n} (W_H^i \setminus W_H^{a+1} \cap W_H^a), a < n \\ W_{gH}^{n,n+1} &= \frac{1}{n} \sum_{i=1}^{n} (W_H^i \cap W_H^n) \end{aligned} \tag{10}$$

$$W_{gH} = W_{gH}^{1,2} \cup W_{gH}^{2,3} \cup \cdots \cup W_{gH}^{n,n+1} \tag{11}$$

However, this method may lead to lower accuracy in a larger model architecture. Take Fig. 6 as an example, the parameters in $W_{gH}^{1,2}$ are only trained on private dataset 1. These parameters impact the convergence of local mode 1 severely, reducing its accuracy, particularly on the Non-IID data. Thus, we also proposed to use the prototype scheme [32] to correct these parameters and enable these

Algorithm 2: FL Aggregation – GlobalTrain

```
Input: model, training settings: $ts$, local_epoch: $le$, start_zero
       round: $s$, latency constraint: $l$, client number: $n$.
Output: local models and their parameters
 1: Initialize the original model and the prototype array $pt$;
 2: Duplicate the model for each client;
 3: for $m \leftarrow 1$ to $n$ do
 4:     Extend $model_m$ if its latency on $client_m$ less than $l$;
 5: end for
 6: Sort models by their complexity in descending order;
 7: for $c \leftarrow 1$ to $Round_{max}$ do
 8:     for $m \leftarrow 1$ to $n$ in parallel do
 9:         Send $model_m$ to the corresponding $client_m$;
10:         $model_m^{'}, pt_m \leftarrow$ LocalTrain($model_m, ts, c, s, le, l, pt$);
11:     end for
12:     if $c \neq Round_{max}$ then
13:         Derive each part of $W_{gH}$ by Eq. (10) and all $model_m^{'}$;
14:         for $m \leftarrow 1$ to $n$ do
15:             $model_m \leftarrow \cup_m^n W_{gH}^{m,m+1}$;
16:         end for
17:         Merge all $pt_m$ and update $pt$ by Eq. (12);
18:     end if
19: end for
```

parameters to learn the pattern in other datasets (*Line 10, 17*). As shown in the circle of Fig. 6, O_m^i is the output of the identical fully-connected layer near the final layer, and it can be used to represent a class by a high-dimension vector. This vector is called the prototype of this class. As shown in Eq. (12), we first aggregate all prototypes by class from all local models at training iteration t. Then, in the next training iteration $t+1$, we add a penalty item between the current prototype of a class and the prototype of this class from previous iterations into the loss function, as illustrated in Eq. (13), where L_1 represents the loss function of the model outputs to labels, L_2 means the loss function of the prototype and β is a pre-defined coefficient. This is the proto-training used in Algorithm 1. In this way, larger models can perceive patterns of all private datasets to improve accuracy.

$$t : proto[l_j]^t = \frac{1}{n}\sum_{i=1}^{n} O_m^i[label == l_j], l_j \in label \tag{12}$$

$$t+1 : Loss = L_1(label, out) + \beta L_2(O_m, proto^t) \tag{13}$$

Training Overhead: The global training adds the proto-corrected scheme. Derived from Eq. (12)-(13), the time complexity of this scheme is $O(IV)$, where I is the total number of training data and V is the dimension of O_m^i. This complexity is not comparable to a traditional aggregation scheme, which is only related to the number of model parameters and clients. But when considering the local training, its complexity is much lower than $O(I\sum_{l=1}^{L} M_l^2 K_l^2 C_{l-1} C_l)$. The model extension introduces overhead to the learning process, but it is intended to improve accuracy for powerful systems. And the model does not belong to the training algorithm, so the model extension is not included in the overhead of the training algorithm.

IV. Experiments

In this section, we first present details of our implementation and the experimental environment, including hardware platforms, benchmark models and datasets. Then we discuss the performance of *Collate*, compared to state-of-the-art methods: Expanding the Reach of Federated Learning (ERFL) [19], Helios [7] and HeteroFL [8]. Their brief is in Section II.

A. Experimental Setup

Hardware Platforms: We collaboratively train models for five distinct devices to evaluate *Collate*. Each device and its inferred model constitute an edge system. These devices are HP ProBook 440 G6, NVIDIA Jetson TX2, NVIDIA Jetson Nano, Raspberry Pi 4B and Samsung Galaxy Noto10. HP ProBook is equipped with an Intel Core i5-8265U CPU with 8GB of memory. TX2 is equipped with an NVIDIA Pascal GPU and 8GB of memory. Nano has an NVIDIA Maxwell GPU and 4GB of memory. Pi owns an ARM Cortex-A72 embedded CPU and 4GB of memory, and Note10 is equipped with a Qualcomm Snapdragon 855 processor and 8GB of memory. Also, *Collate* does not limit the number of clients.

Datasets and Models: The datasets used for the benchmark are MNIST, CIFAR-10, CIFAR-100 and Human Activity Recognition (HAR) [33], which are widely used in evaluating Federated Learning [6]–[10], [19]–[21]. The first three are image classification datasets, which are popular computer vision applications. And HAR is an attractive feature for smartphones using data collected from different types of onboard sensors. We split the training data of each dataset into five parts, and each device owns a part and keeps it private. The trained model is evaluated on all test data of the corresponding dataset. For the generation of IID data, we randomly split the training data, and each class in a dataset exists on all five devices. And for the generation of Non-IID data, we force each class of a dataset to be divided into at most two parts, thus, a class only exists on at most two devices. The used models are LeNet, AlexNet, ResNet-18 and IMU [34], and they are trained on MNIST, CIFAR-10, CIFAR-100 and HAR, respectively.

Implementation Details: The proposed method is implemented using the Pytorch library [35]. We set the local training epoch to 6, and the communication rounds are 20, 30, 40 and 100 for LeNet, IMU, AlexNet and ResNet-18, respectively. And for all datasets, the optimizer is SGD with a momentum of 0.9 and no weight decay. The learning rates are set to 0.05, 0.003, 0.001 and 0.01 for LeNet, IMU, AlexNet and ResNet-18, respectively, which are widely used for these models. To exhibit the superiority of *Collate* under the latency constraint, we re-implement ERFL [19], Helios [7] and HeteroFL [8] according to their papers or open-source codes. We use the latency predictor for each method and modify them to guarantee the latency constraint of each local model. And we use the same training settings for all methods.

B. Accuracy Evaluation

Our method can also learn models with pre-defined complexities, reflected by changing the $Shrink\_ratio$ to a constant in local training (*Line 9* of Algorithm 1). Due to HeteroFL [8] using pre-defined complexities to learn models, their method can easily include the model extension. We follow their work to set five discrete complexity levels of 1.5, 1.25, 1.0, 0.5, 0.25 to evaluate *Collate* and HeteroFL [8]. However, ERFL [19] and Helios [7] can only shrink the model size, thus, we use 1.0, 0.5, 0.25 complexity levels for these two works.

Results on IID data: The top five in Fig. 7 show that *Collate* achieves the highest accuracy compared to HeteroFL [8], Helios [7], and ERFL [19] under different complexity levels for the MNIST dataset. The original method is the traditional identical FL [13], and it obtains only one global model with the same complexity (1.0).

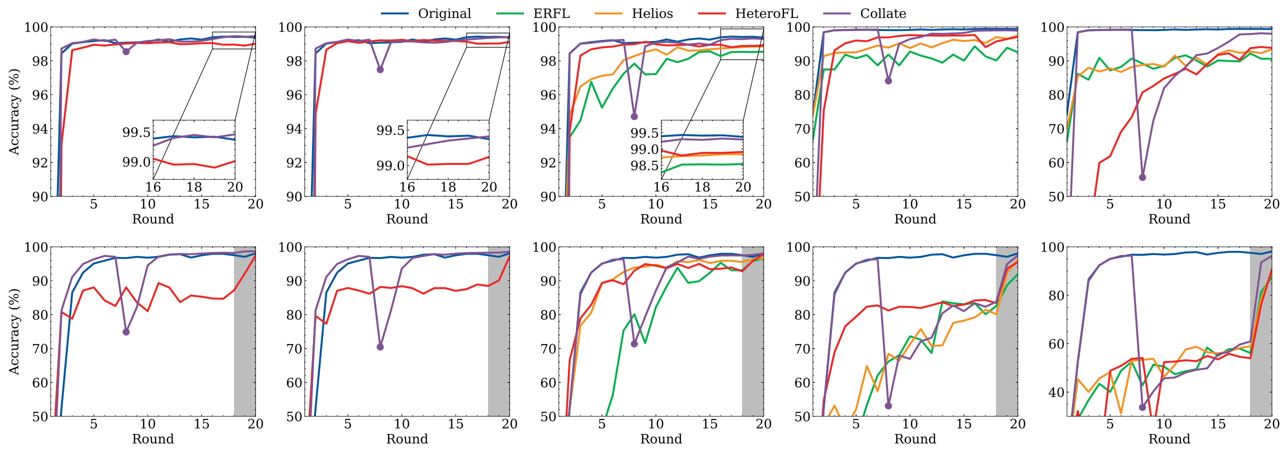
Fig. 7. The test accuracy of LeNet on MNIST under different complexity levels (From left to right are 1.5, 1.25, 1.0, 0.5, 0.25). The top five show results on IID data and the bottom five are on Non-IID data. • points to the communication round that starts the zeroizing-recovering process of *Collate*.

TABLE I
THE TEST ACCURACY ON HAR UNDER DIFFERENT COMPLEXITY LEVELS.

Data DIST	Method	Accuracy (%) of Different Complexities				
		C-1.50x	C-1.25x	C-1.00x	C-0.50x	C-0.25x
IID	**Original**	N.A.	N.A.	94.53	N.A.	N.A.
	ERFL [19]	N.A.	N.A.	93.87	89.64	81.53
	Helios [7]	N.A.	N.A.	94.33	91.28	88.72
	HeteroFL [8]	93.57	94.60	**94.81**	93.63	90.17
	Collate	**96.40**	**96.33**	94.40	**94.26**	**91.68**
Non-IID	**Original**	N.A.	N.A.	76.33	N.A.	N.A.
	ERFL [19]	N.A.	N.A.	74.21	71.26	66.05
	Helios [7]	N.A.	N.A.	75.16	71.63	67.87
	HeteroFL [8]	75.03	75.70	75.67	73.68	69.32
	Collate	**77.54**	**78.15**	**76.38**	**75.57**	**71.53**

TABLE II
THE TEST ACCURACY ON CIFAR-10 UNDER THE LATENCY CONSTRAINT.

Data DIST	Method	Accuracy (%) of Local Models				
		ProBook	Tx2	Nano	Pi	Note10
IID	**Original**	82.72	82.72	**82.72**	N.A.	N.A.
	ERFL [19]	76.53	76.53	76.53	75.80	76.28
	Helios [7]	81.24	81.24	81.24	79.30	79.42
	HeteroFL [8]	82.20 (E)	81.11 (E)	82.23	74.35	78.66
	Collate	**83.74 (E)**	**83.93 (E)**	81.82	**80.40**	**81.72**
Non-IID	**Original**	80.25	80.25	80.25	N.A.	N.A.
	ERFL [19]	75.80	75.80	75.80	63.06	75.13
	Helios [7]	79.82	79.82	79.82	69.62	76.28
	HeteroFL [8]	80.15 (E)	79.02 (E)	80.13	69.15	76.86
	Collate	**81.45 (E)**	**81.11 (E)**	**80.66**	**74.83**	**78.29**

We put its same curve in these five sub-figures for easy comparison. Compared with HeteroFL [8] on model extension, *Collate* with the proto-corrected scheme can achieve higher accuracy than the original method, while the accuracy of HeteroFL [8] is lower than the original method. Compared with Helios [7] and ERFL [19], *Collate* can achieve the highest accuracy in the shrunk models through the well-designed Zeroizing-Recovering process. Table I (IID part) presents that on the HAR dataset, *Collate* can also perform well in most cases. All highest accuracy is marked in **boldface**. The accuracy of our extended models can outperform the original one by about 2% and our shrunk models achieve the lowest accuracy loss.

Results on Non-IID data: The bottom five in Fig. 7 demonstrate that *Collate* still performs well on the Non-IID data of the MNIST dataset. Different from IID data, for each method and each dataset, we use the last 10% of the total communication rounds to fine-tune each local model in a traditional FL way [13]. This is because final models are generated after local training, and aggregation updates zeroized weights into non-zero in all methods (except HeretoFL, but for fairness, we also fine-tune HeteroFL). Updating only by the local dataset impacts its accuracy on all test sets, especially under Non-IID data. Fig. 7 also shows that some of our models perform well without fine-tuning by the proto-corrected scheme. Table I (Non-IID part) shows that on the HAR dataset, *Collate* can still achieve the highest accuracy.

C. Latency-Accuracy Evaluation

To demonstrate the effectiveness of our method for multiple latency-critical edge systems, we evaluate the accuracy of *Collate* and state-of-the-art heterogeneous FL methods under a pre-defined latency constraint. As Helios used Nvidia Jetson Nano as the non-straggler [7], we also pick the original model inference latency on Jetson Nano as the constraint, which means we adjust all local models to achieve a similar latency as that of the original corresponding model on Jetson Nano.

Table II presents that compared with ERFL [19], Helios [7] and HeteroFL [8], *Collate* can achieve the best accuracy for almost all the five platforms on IID or Non-IID data of the CIFAR-10 dataset. As HeteroFL [8] and our method can extend models to larger than the original ones, while others can only shrink models, we use E-flag to represent model extension. Additionally, each local model in Table II can meet the latency constraint (38 ms) and achieve a latency in the range of $36.8\pm0.7ms$. For methods without extension, the latency of their models is lower than the latency constraint on powerful devices. Compared to the original model, for the IID data, we extend the model on HP ProBook from about 23.0 ms and improve the accuracy by 1.02%. Also, the latency of the original model on Note10 is around 82.6 ms, and we reduce its latency by about 54.0% with only a 1.00% accuracy drop. For the Non-IID data, since it is more difficult to learn the pattern of the dataset, the benefit of *Collate* is less than that of IID data, but our method still outperforms other methods.

Table III shows the evaluation results of *Collate* on the CIFAR-100

TABLE III
THE TEST ACCURACY ON CIFAR-100 UNDER THE LATENCY CONSTRAINT.

Data DIST	**Method**	**Accuracy (%) of Local Models**				
		ProBook	**Tx2**	**Nano**	**Pi**	**Note10**
IID	**Original**	70.25	70.25	70.25	N.A.	N.A.
	ERFL [19]	69.78	69.78	69.78	68.30	69.82
	Helios [7]	70.14	70.14	70.14	68.30	70.23
	HeteroFL [8]	68.70 (E)	70.25 (E)	70.35	67.25	70.48
	Collate	**70.68 (E)**	**72.27 (E)**	**70.69**	**68.54**	**70.87**

dataset. Each local model in Table III is under a latency constraint of about 70 ms and they achieve a latency in the range of $67.8 \pm 1.8ms$. Among these models, *Collate* obtains the best accuracy under all conditions. Compared to the original model, we extend it on Jetson TX2 from about 27.4 ms and obtain a 2.02% accuracy improvement. Also, the latency of the original model on Pi is around 152.6 ms, and we reduce its latency by about 54.1% with only a 1.71% accuracy drop. Specifically, from Table II-III, we can conclude that, compared to other methods [7], [8], [19], *Collate* improves the accuracy by 1.96% on average for extended models, and our shrunk models can obtain a 3.09% accuracy improvement on average.

V. CONCLUSION

In this work, we have presented a collaborative neural network learning framework for multiple latency-critical edge systems to directly satisfy the latency constraint of DNNs on their devices. The collaboration augmented the training data for each system to improve accuracy while also protecting their data privacy. Our framework can satisfy the latency constraints of each system with one training process. It contained two components: a dynamic zeroizing-recovering algorithm and a proto-corrected aggregation scheme. The dynamic algorithm can extract optimal model architectures to achieve less accuracy drop under a large latency reduction. Our aggregation scheme improved the accuracy of extended models. We also included the latency predictor and model extension to guide and optimize this framework. Experiments verified that our framework has obtained remarkable improvements.

REFERENCES

[1] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys*, 2018.

[2] X. Zhu, C. Vondrick, C. C. Fowlkes, and D. Ramanan, "Do we need more training data?" *International Journal of Computer Vision*, 2016.

[3] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[4] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big. little multicore processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2254–2267, 2019.

[5] Y. Cheng, Y. Liu, T. Chen, and Q. Yang, "Federated learning for privacy-preserving ai," *Communications of the ACM*, 2020.

[6] J. Zhang, S. Guo, X. Ma, H. Wang, W. Xu, and F. Wu, "Parameterized knowledge transfer for personalized federated learning," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[7] Z. Xu, F. Yu, J. Xiong, and X. Chen, "Helios: heterogeneity-aware federated learning with dynamically balanced collaboration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.

[8] E. Diao, J. Ding, and V. Tarokh, "Heterofl: Computation and communication efficient federated learning for heterogeneous clients," *arXiv preprint arXiv:2010.01264*, 2020.

[9] D. Li and J. Wang, "Fedmd: Heterogenous federated learning via model distillation," *arXiv preprint arXiv:1910.03581*, 2019.

[10] Y. Tan, G. Long, L. Liu, T. Zhou, Q. Lu, J. Jiang, and C. Zhang, "Fedproto: Federated prototype learning across heterogeneous clients," in *AAAI Conference on Artificial Intelligence*, vol. 1, 2022.

[11] B. Yuan, C. R. Wolfe, C. Dun, Y. Tang, A. Kyrillidis, and C. M. Jermaine, "Distributed learning of deep neural networks using independent subnet training," *arXiv preprint arXiv:1910.02120*, 2019.

[12] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[13] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017.

[14] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[15] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018.

[16] K. Wang, R. Mathews, C. Kiddon, H. Eichner, F. Beaufays, and D. Ramage, "Federated evaluation of on-device personalization," *arXiv preprint arXiv:1910.10252*, 2019.

[17] Y. Jiang, J. Konečnỳ, K. Rush, and S. Kannan, "Improving federated learning personalization via model agnostic meta learning," *arXiv preprint arXiv:1909.12488*, 2019.

[18] M. Khodak, M.-F. F. Balcan, and A. S. Talwalkar, "Adaptive gradient-based meta-learning methods," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[19] S. Caldas, J. Konečny, H. B. McMahan, and A. Talwalkar, "Expanding the reach of federated learning by reducing client resource requirements," *arXiv preprint arXiv:1812.07210*, 2018.

[20] A. Li, J. Sun, P. Li, Y. Pu, H. Li, and Y. Chen, "Hermes: an efficient federated learning framework for heterogeneous mobile clients," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 420–437.

[21] E. Jeong *et al.*, "Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data," *arXiv preprint arXiv:1811.11479*, 2018.

[22] D. S. Wibowo and S. J. Fong, "Mobile blockchain-empowered federated learning: Current situation and further prospect," in *2021 Third International Conference on Blockchain Computing and Applications (BCCA)*. IEEE, 2021, pp. 19–25.

[23] T.-J. Yang *et al.*, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *ECCV*, 2018.

[24] X. Dai *et al.*, "Chamnet: Towards efficient network design through platform-aware model adaptation," in *CVPR*, 2019.

[25] A. Gholami *et al.*, "Squeezenext: Hardware-aware neural network design," in *CVPRW*, 2018.

[26] B. Wu *et al.*, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *CVPR*, 2019.

[27] D. Justus *et al.*, "Predicting the computational cost of deep learning models," in *2018 IEEE international conference on big data (Big Data)*. IEEE, 2018, pp. 3873–3882.

[28] S. Huai *et al.*, "Zerobn: Learning compact neural networks for latency-critical edge systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 151–156.

[29] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural Networks*, 1992.

[30] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi, "Morphnet: Fast & simple resource-constrained structure learning of deep networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1586–1595.

[31] G. Goh, "Why momentum really works," *Distill*, 2017.

[32] W. Xu, Y. Xian, J. Wang, B. Schiele, and Z. Akata, "Attribute prototype network for zero-shot learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 969–21 980, 2020.

[33] D. Anguita, A. Ghio, L. Oneto, X. Parra Perez, and J. L. Reyes Ortiz, "A public domain dataset for human activity recognition using smartphones," in *Proceedings of the 21th international European symposium on artificial neural networks, computational intelligence and machine learning*, 2013, pp. 437–442.

[34] F. Moya Rueda *et al.*, "Convolutional neural networks for human activity recognition using body-worn sensors," in *Informatics*, vol. 5, no. 2. Multidisciplinary Digital Publishing Institute, 2018, p. 26.

[35] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019.