Context-Aware Compilation of DNN Training Pipelines across Edge and Cloud

DIXI YAO, Shanghai JiaoTong University, China
LIYAO XIANG*, Shanghai JiaoTong University, China
ZIFAN WANG, Shanghai JiaoTong University, China
JIAYU XU, Shanghai JiaoTong University, China
CHAO LI, Shanghai JiaoTong University, China
XINBING WANG, Shanghai JiaoTong University, China

Empowered by machine learning, edge devices including smartphones, wearable, and IoT devices have become growingly intelligent, raising conflicts with the limited resource. On-device model personalization is particularly hard as training models on edge devices is highly resource-intensive. In this work, we propose a novel training pipeline across the edge and the cloud, by taking advantage of the powerful cloud while keeping data local at the edge. Highlights of the design incorporate the parallel execution enabled by our feature replay, reduced communication cost by our error-feedback feature compression, as well as the context-aware deployment decision engine. Working as an integrated system, the proposed pipeline training framework not only significantly speeds up training, but also incurs little accuracy loss or additional memory/energy overhead. We test our system in a variety of settings including WiFi, 5G, household IoT, and on different training tasks such as image/text classification, image generation, to demonstrate its advantage over the state-of-the-art. Experimental results show that our system not only adapts well to, but also draws on the varying contexts, delivering a practical and efficient solution to edge-cloud model training.

CCS Concepts: • **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → *Neural networks*; • **Security and privacy** → Privacy protections.

Additional Key Words and Phrases: edge computing, neural networks, machine learning

ACM Reference Format:
Dixi Yao, Liyao Xiang, Zifan Wang, Jiayu Xu, Chao Li, and Xinbing Wang. 2021. Context-Aware Compilation of DNN Training Pipelines across Edge and Cloud. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 4, Article 188 (December 2021), 27 pages. https://doi.org/10.1145/3494981

1 INTRODUCTION

A wide variety of applications have begun to embrace the trend of intelligent edge — integrating powerful machine learning technologies with the portable, mobile edge. The edge could be smartphones, wearable gadgets, or IoT devices, and the applications range from personal recommendations [55], health data analysis [6], to

*Liyao Xiang is the corresponding author (xiangliyao08@sjtu.edu.cn).

Authors' addresses: Dixi Yao, Shanghai JiaoTong University, China, Jimmyyao18@sjtu.edu.cn; Liyao Xiang, Shanghai JiaoTong University, Shanghai, China, xiangliyao08@sjtu.edu.cn; Zifan Wang, Shanghai JiaoTong University, Shanghai, China, wzf184@sjtu.edu.cn; Jiayu Xu, Shanghai JiaoTong University, Shanghai, China, bilker@sjtu.edu.cn; Chao Li, Shanghai JiaoTong University, Shanghai, China, lichao.1024@sjtu.edu.cn@sjtu.edu.cn; Xinbing Wang, Shanghai JiaoTong University, Shanghai, China, xwang8@sjtu.edu.cn.

2474-9567/2021/12-ART188 $15.00
https://doi.org/10.1145/3494981

natural language processes [4, 34, 54] and robotics [38]. A salient characteristic of the applications is that the data is expected to be processed locally while providing personalized models of high quality. However, such a feature contradicts the resource-stringent edge: machine learning algorithms typically demand great computational power, usually far beyond the capability of a thin piece of device.

Various on-device distributed training frameworks are proposed to address the issue: *split learning* enables the edge and cloud to collaboratively train a model without sharing any raw data; *transfer learning* fine-tunes a pre-trained model on a device to migrate knowledge to a customized context; *cross-silo federated learning* aligns attributes from different devices for the same entity to learn a shared model. These frameworks address the requirement of personalized models but also pose running time challenges. In user cases such as location prediction [55], elderly care robots [38], input words prediction [4, 34], etc., users' data are frequently updated and the model needs to fast adapt to new contexts through efficient training. For example, the input suggestion engine on the smartphone would have parts of the recommendation model constantly fine-tuned on the user's private inputs.

On-device model personalization is particularly hard as training relies on repeated access to local data and iterative computation procedures, while simply resorting to the powerful cloud backend does not solve the problem. Notably, the edge hardware has gone through rapid development while today's 5G network achieves a maximal 1Gbps throughput. The computation capability of the cloud is also growing aggressively with the advance of hardware and system architecture. Hence it is beneficial to take the remote cloud as a viable remedy to the thin edge. It is possible to orchestrate the edge, the network, and the cloud for efficient model personalization. However, the conventional edge-cloud collaborative learning mostly handles inference tasks [22]: the model stub on the edge forwards the input to obtain the intermediate-layer features which are sent to the cloud, whereas the cloud completes inference. But a training task contains not only forward, but also backward propagation. A single forward-backward loop would involve data transfer back and forth between the edge and cloud, rendering the two waiting for each other. More importantly, this has to be done iteratively in the dynamic networking environment.

In this work, we introduce an efficient, practical training framework across the edge and cloud by breaking the serialization lock in the forward-backward loop. We decouple the forward and backward steps so that the two parts can run in parallel. To enable parallel execution, we reuse previous forwarded data to replay on the current weights, and thus the backward step does not have to wait for the error gradients of the most recent inputs. With the technique as a backbone, we further parallelize the computation and the data transfer so that four modules run simultaneously without the device or cloud sitting idle. To further optimize the communication, we propose an error-feedback feature compression method, to train on quantized features with little accuracy loss.

We observe our parallel execution and compression leave much design space to speed up training, due to the incomparable computational power between the edge and cloud. The partition layer of the model, the parallelism factor, and the per-layer bit-width are critical to training latency and model accuracy. The actual context, including the platform, model, task, and network connectivity also play a part in optimizing performance. Hence we build a dynamic decision engine to adaptively schedule deployment. Whenever the context alters, the engine re-calculates the optimal strategy and triggers re-deployment across the edge and the cloud. To verify the effectiveness, we implement and test our system in a variety of settings such as WiFi, 4G, 5G, household IoT, in split learning, transfer learning, and cross-silo federated learning respectively.

Highlights of our contribution are as follows. First, we are among the first to propose an edge-cloud training pipeline by exploiting parallelism across the edge and cloud. With scheduled feature replay and error-feedback compression, we speed up model personalization almost without accuracy degradation. Second, we design a context-aware decision engine on the edge to adaptively arrange parallel execution and compression. It embraces flexibility to optimize latency and accuracy performance under a wide range of circumstances. Finally, our system

is verified in a variety of settings and the experimental evidence supports that the proposed framework has superior performance over the state-of-the-art, demonstrating significant application potential.

2 RELATED WORK

The most related works to ours fall into the following categories.

2.1 Context-Aware Edge-Cloud Computing

Previous works have shown a wide range of possibilities to perform joint learning between the edge and cloud, such as [10, 14, 17, 18, 22, 29, 46, 51]. Han *et al.* [14] systematically traded off neural network classification accuracy for resource use in the multi-programmed, streaming setting with an optimization-based heuristic scheduling algorithm. Teerapittayanon *et al.* [46] proposed to map sections of a model onto a distributed computing hierarchy while maximizing the utility of the extracted features. Lin *et al.* [29] extended the problem to contain multiple edges, fogs, as well as cloud devices and solved it with a genetic algorithm. Besides heuristic algorithms, some works proposed deterministic algorithms to find the optimal partition policy. Kang *et al.* [22] managed to find the optimal partition for chain-like neural networks. Hu *et al.* [17] turned the problem into a min-cut problem and found the optimal partition for models represented by Directed Acyclic Graphs (DAG). Similarly, by modeling the model structure as a DAG, JointDNN [10] provided optimization formulations at layer granularity as the shortest path problem on a DAG.

Recent works such as [1, 18, 27, 51] have shifted the paradigm to scenarios of finer granularity and fast-changing dynamics, due to the adaptability of the system to varied contexts is critical for efficiency. Clio [18] presented a novel approach to split machine learning models between an IoT device and cloud progressively adapting to wireless dynamics. Wang *et al.* [51] searched for transformation on a model tree according to the constantly-changing network environment. DynO [1] exploited the variability of precision needs in different parts of a model in adapting inference to the execution environment, and presented a novel quantization method to speed up communication. SPINN [27] optimized its scheduler with an early exit policy and performs accuracy-aware lossy compression. We also propose to optimize the communication in our system with an error-feedback quantization method adapting to a dynamic training context. The wider time window and iterative computation is a unique challenge in our scenario.

2.2 On-device Model Personalization

A recent trend in machine learning services is to provide customized models on a per-user basis, rather than a suitable-for-all model. Where-to-go-next [55] used a Spatio-temporal gated network to model personalized sequential patterns with user-POI interactions and locations POIs. Yoon *et al.* [54] tuned a general language model into a personalized model with a small amount of user data while preventing transferring the data out of the device. Saunders *et al.* [38] taught a robot to meet the changing needs of the elderly over time. Vallo *et al.* [47] proposed a data-driven modeling approach to capture the lane change decision behavior of human drivers, which allows a vehicle to autonomously change lanes in a safe but personalized fashion. Sim *et al.* [42] trained personalized end-to-end speech recognition models on mobile devices. Lin *et al.* [30] supported the real-time personalization of human activity recognition models in the training stage. Lange *et al.* [26] proposed a dual user-adaptation framework to address the task of unsupervised model personalization while maintaining privacy and scalability. A critical issue in the presented cases such as POI prediction, vehicle lane prediction, elderly care robots, is the lengthy training period, which would cause personalized models out-of-date and fail to meet users' demands. For example, to obtain high-quality transcriptions for on-device speech recognition, Sim *et al.* [42] reduced the training time from thousands of hours to only 1.5 hours per user. For a case of predicting possible dining locations, an expected model finetuning latency would be the period of taking a meal like an hour or

less [55]. We address the problem in this paper by fully tapping into the computational power on the edge while developing an adaptive decision engine to keep the overall latency low.

2.3 Distributed DNN Training

As the deep learning system scales up, a variety of distributed deep learning frameworks have been built [7, 12, 39, 53] to explore the parallelism in data or model. Gpipe [19] utilized the batch-splitting algorithm to let each distributed device run micro-batches in parallel. On-device transfer learning [44] fine-tunes a model pre-trained elsewhere to facilitate convergence on the device. Split learning [36, 49, 50] allows various configurations of collaborative parties to jointly train a machine learning model without raw data sharing [49]. A typical way is to 'split' a neural network and place parts on different devices which send the computational results of each part to one another. Cross-silo federated learning [53] is a model training framework where data attributes are dispersed on different devices. These frameworks can be found in multi-headed SplitNN [37], CTR prediction [52], B2B (business-to-business) model [53], and VQA [31]. Since the data flow of camera images, transaction records, the user clicks continuously arrive, fast training and prediction are also desired. We show our system can well support these on-device distributed training frameworks.

3 PRELIMINARIES

In this section, we introduce the background for ease of understanding our work.

3.1 Breaking Backward Locking

Forward and backward compose two main phases in training a neural network. In the forward loop, the input is fed layer by layer to the model to produce predictions, while the backward loop passes the error gradients sequentially from the loss to the input layer to update the parameters of each layer. Assume that a network is partitioned into K modules. L_k represents the kth module ($1 \leq k \leq K$). We use w_k^t to denote the weights of module k at timestamp t. Upon finishing the forwarding on module k and generating output feature h_k^t, we normally need to wait for the error gradient δ_k^t to be sent back from the loss. Only when the modules following k finish their backward propagation, can the k-th module receive the error gradient, and update its weights. As a result, module k needs to wait for $K-k$ time slots between its forward and backward loops.

Unlike the common backpropagation algorithm, feature replay [21] can decouple the forward and backward loop with a novel parallel-objective formulation. At timestamp t, module k has the error gradient δ_k^{t-K+k} corresponding to the forwarding at timestamp $t-K+k$. In the feature replay algorithm, the module first replays the input feature at $t-K+k$ on module k and gets replay feature $\tilde{h_k^t} = F(h_{k-1}^{t-K+k}, w_k^t)$. Then the algorithm uses δ_k^{t-K+k} to approximate δ_k^t and performs the backpropagation:

$$g_k^t = \frac{\partial \tilde{h_k^t}}{\partial w_k^t} \times \delta_k^{t-K+k}, \quad \text{and} \quad \delta_{k-1}^{t-K+k} = \frac{\partial \tilde{h_k^t}}{\partial \tilde{h_{k-1}^t}} \times \delta_k^{t-K+k}. \tag{1}$$

Then module k sends δ_{k-1}^{t-K+k} to module $k-1$ and module $k-1$ apply the same method to update its weights. Since the error gradients, each module uses to update its weights are generated at different timestamps, all modules could conduct backpropagation in parallel. Training with feature replay is guaranteed to converge to critical points according to the conclusion of [21].

Although the feature replay algorithm can well resolve the backward locking problem, we cannot directly apply the method to the edge-cloud setting. First, we only have two 'modules': the cloud and the edge, *i.e.,* $K = 2$, which cannot fully utilize the feature replay algorithm to speed up backpropagation. Moreover, in the feature replay algorithm, modules are run on GPUs with comparable computational power so that module k expects to obtain error δ_k^{t-K+k} at timestamp t. However, in the edge-cloud infrastructure, due to the big resource gap

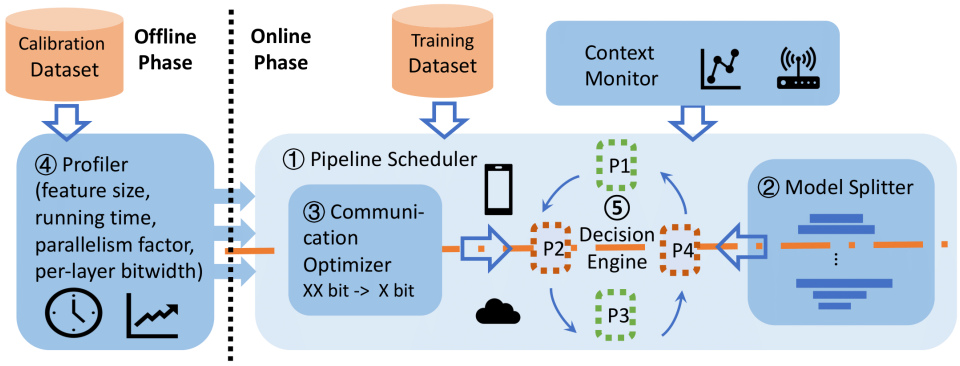


Fig. 1. System overview.

between the thin edge device and the cloud, module 1 requires a significantly longer time to wait for the error δ_1^{t-1} of previous iterations, which is almost the same as serialized computation.

3.2 Feature Compression

Feature compression is commonly used to reduce the communication overhead between the edge and the cloud [1, 5, 10, 27]. There are lossy and lossless compression methods. The lossy compression method has a great impact on the convergence performance and the final accuracy. Both [5] and [41] require compression decoders to compensate for the losses. When the deployment of the model changes across the edge and the cloud, the previously trained decoder would fail on the newly compressed features. Hence their method does not apply to the highly dynamic scenario as ours. On the other hand, the lossless compression method lacks flexibility, as it can only be used at some fixed partition point of the model. For example, in [10], the PNG compression method can only be applied to layer outputs encoded as images. From the hardware perspective, [3] presents a hardware-friendly lossless compression scheme for feature maps of convolutional neural networks.

In this work, we simply choose quantization as the compression technique. As in [5], if we choose an n-bit uniform quantization for the intermediate-layer feature $h \in \mathbb{R}^{H \times W \times N}$ (height H, width W, and channel numbers N), we would send a quantized feature $C(h)$ in place of h:

$$C(h) = \left\lfloor \frac{h - min(h)}{max(h) - min(h)} \cdot (2^n - 1) \right\rceil. \tag{2}$$

Other compression techniques such as LZ4, GZip, or input-dependent quantization [1] can also be applied but we use n-bit quantization for simplicity. Using $C(h)$ in place of h would result in errors that would be accumulated in training and finally lead to accuracy loss. Hence it is a practical challenge to reduce the quantization error.

4 SYSTEM DESIGN

In this section, we first give an architecture overview of our proposed system and introduce each part one by one. As shown in Fig. 1, there are two phases in the design. During the offline phase, we use a calibration set that is sampled from the validation set to profile the model to be trained. On the targeted device, we only need to run the calibration set through the model once to record its feature size at each layer, and the running time until reaching each layer. Given the model's tolerance for a certain accuracy drop, we also train the model by the calibration set on the server to obtain the maximum parallelism factor, as well as the minimum per-layer bit

width. The feature size, running time, parallelism factor, and per-layer bit width will be used in the online phase for model configuration.

At the online phase, our pipeline scheduler iteratively takes a batch of training data, the current network condition, and the profiling results to produce a real-time configuration. The core of the pipeline scheduler lies in a generalized feature replay algorithm to parallelize the execution across the edge and the cloud. It consists of four execution queues to enable a variety of configuration choices for training modules. Depending on the scheduling outcome, the model splitter divides the neural network into two partitions, and the communication optimizer compresses features in the forwarding loop before sending them to the cloud, while compensating for the compression error in the backward loop. By minimizing the training latency of the cost model, our decision engine dynamically searches for the optimal decision for deployment. In the following, we give a detailed description of each part.

4.1 Pipeline Scheduler

A naive machine learning training pipeline across the edge and cloud is displayed on the left side of Fig. 2. For each batch of training data, the edge device feeds them to the local neural network and outputs features to send to the cloud. The edge needs to wait until the cloud finishes the backward loop and returns the error gradient, with which the edge finishes the rest of the backward loop. The edge and cloud have to wait for each other to proceed.

We break the lock by modularizing the training procedure, with each module running individually and in parallel. We use t to denote the sequence number of the input batch ($t = 0, 1, \cdots, T$) as well as the iteration number. A single iteration is divided into four modules, and we take the t-th iteration as an example: the first module is the backward procedure of batch $t-1$, followed by the forwarding of batch t on the edge (marked as P_1 in the right part of Fig. 2). The second module is the step of uploading the batch t feature to the cloud (P_2). The third module contains the forwarding of the batch t feature as well as its backward step (P_3). The fourth module is the edge downloading error gradient of t from the cloud (P_4). A naive way of training would result in resource waste: when P_1 is running, the network card and the cloud server go idle; when P_3 is executing, the network card and the edge have to wait for the returned results.

Hence we let the four modules run 'independently' without tight coupling. But each module still serves as a parallelization atom, *i.e.,* no steps can be executed in parallel within each module. To enable parallelism across modules, we specifically design four queues to store requests of each module, and a history queue to store history information. Q_1 and Q_4 are on the edge whereas Q_2 and Q_3 are on the cloud. After the edge has forwarded batch t, it inserts its feature into Q_1 waiting to be sent, copies it to the history queue, and then checks if Q_4 is empty. If not, the edge read the error gradient from Q_4 to update the local model partition. Note that the error gradient is of batch $t' < t$ from a previous iteration. Then the edge goes to the forwarding step of batch $t+1$.

As long as Q_1 is non-empty, the edge uploads the feature to the cloud where the feature is stored in Q_2. The cloud server is driven by the event that Q_2 is not empty, and the cloud takes the feature out to perform forwarding and the backward step. The produced error gradient is put into Q_3 and triggers the sending event to send the error gradient back to the edge. The edge receives the error gradient and stores it in Q_4. Due to the FIFO property of the queues, the sequence of batches would remain in order. The four queues work like shared memories, stitching the four modules together.

4.1.1 Feature Replay. One key problem to the aforementioned training pipeline is such that when the edge takes out the error gradients $\delta_c^{t'}$ of batch t' ($t' \leq t$, c represents the error gradients returned from the cloud), the error gradients correspond to the input from previous iteration t', which is misaligned with the forwarded feature h^t of batch t. To allow parallel computation, we manage to use $\delta_c^{t'}$ to update local weights from w^t to w^{t+1}.

We first show the 'correct' but 'invalid' computation of the gradient, and then illustrate how to approximate g^t. Since the error gradient received is for a previous batch t', the gradient update is actually on batch t'. Hence

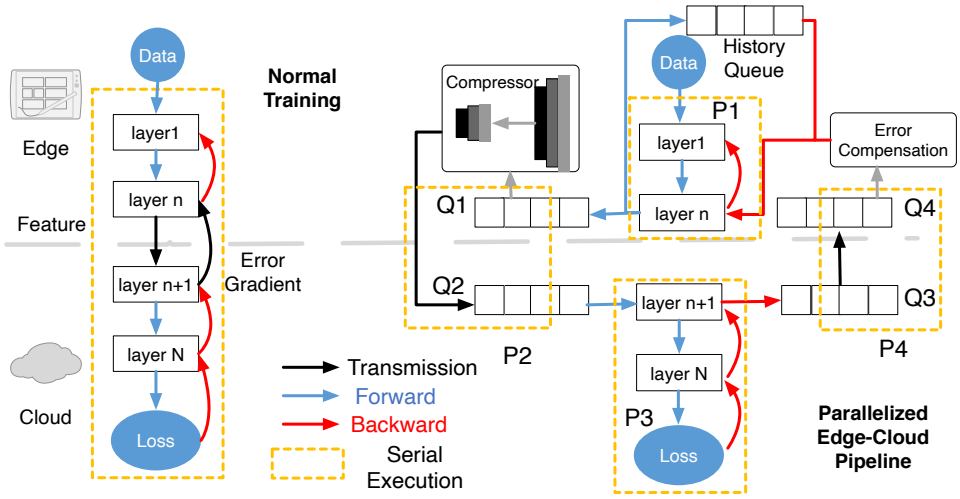


Fig. 2. The normal edge-cloud training (left) and our proposed parallelized edge-cloud training pipeline (right).

the 'correct' gradient is $g^{t'} = \frac{\partial h^{t'}}{\partial w^{t'}} \times \delta_c^{t'}$. However such a gradient cannot update the current weights w^t. To update w^t, we need a forwarded feature on w^t. Thus we replay previous batch $x^{t'}$ on the current parameter w^t to compute the replay feature $\tilde{h}^t = F(x^{t'}, w^t)$, where F is the forwarding network on the edge. Therefore, we update w^t by the following g^t:

$$g^t = \frac{\partial \tilde{h}^t}{\partial w^t} \times \delta_c^{t'} = \frac{\partial F(x^{t'}, w^t)}{\partial w^t} \times \delta_c^{t'}. \tag{3}$$

Note that the only difference between $g^{t'}$ and g^t is that we use the current weights w^t instead of the previous weights $w^{t'}$. It has been proved in [21] that

$$\sum_{i=t'}^{t} \mathbb{E}||\nabla F(w^i)||_2^2 \leq \frac{F(w^{t'}) - F(w_*)}{\sigma\gamma} + \frac{\gamma LM}{2\sigma}(t - t'), \tag{4}$$

where γ, σ, L, M in Eq. 4 are constant and w_* is the optimal solution. As a result, the difference between w^t and $w^{t'}$ is bounded by $\eta\frac{\gamma LM}{2\sigma}(t - t')$. When t is close to t', $w^t \approx w^{t'}$, and thus Eq. 3 represents a valid gradient which allows updating the gradient of the network on the edge in a backward manner. For example, to compute the gradient of the $l-1$-th layer, we perform

$$g_{l-1}^t = \frac{\partial \tilde{h}_l^t}{\partial w^t} \times \delta_l^{t'}, \quad \text{and} \quad \delta_{l-1}^t = = \frac{\partial \tilde{h}_l^t}{\partial \tilde{h}_{l-1}^t} \times \delta_l^{t'}. \tag{5}$$

In the following, we show the weight distribution gaps at t and t' at different training phases through empirical experiments.

4.1.2 Weight Distributions. To understand the weight distribution shift, we measure the weight changing rate at different Ks for the following models: ResNet50 on CIFAR10, Chair on 3DChair, and BI-LSTM on IMDB. The training hyperparameters are default to settings in Table 1. The weight changing rate is defined as:

$$\left| \frac{\|w^t\|_1 - \|w^{t-K}\|_1}{\|w^{t-K}\|_1} \right| \times 100\%. \tag{6}$$

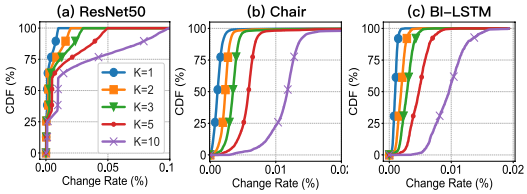


Fig. 3. Cumulative distribution function of the relative weights changing rate over Ks throughout the training process. The smaller the K, the less the weight change.

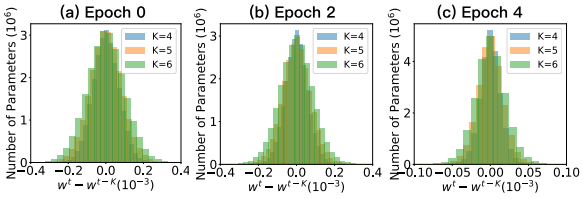


Fig. 4. Weight difference distribution of transferring ResNet50 from ImageNet to CIFAR10 at different phases of the training process. Towards the end of training, weights vary less by iterations.

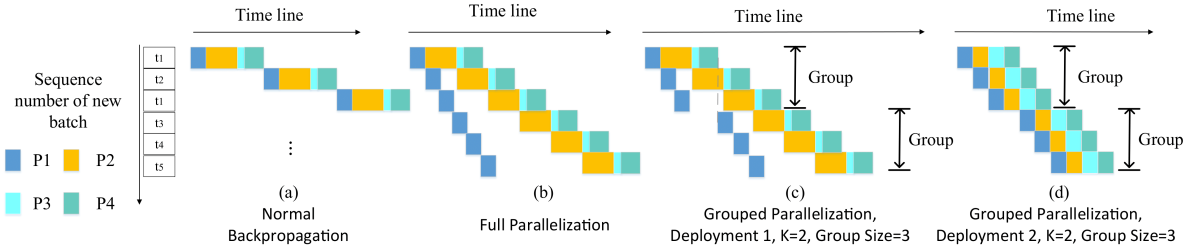


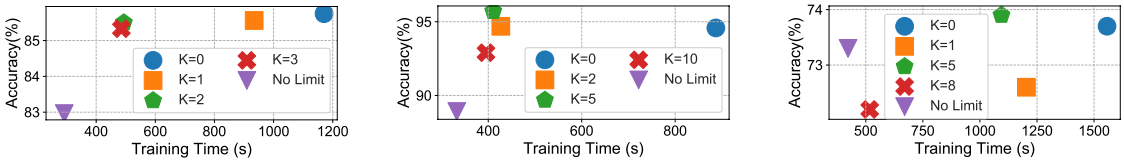
Fig. 5. Training pipelines of normal backpropagation, full parallelization, and grouped parallelization.

It indicates the relative change of the weights' l_1 norm over K iterations. Fig. 3 shows the cumulative distribution function (CDF) of weights changing rate for different models throughout the training process. Most of the weights change remains negligible when K is less than 5.

To observe how the weight distribution shift varies across training phases, we measure the distribution of $w^t - w^{t-K}$ in a randomly selected iteration at the early, middle, and rear stages of training respectively in Fig 4. It is a transfer learning to CIFAR10 with pre-trained ResNet50 on ImageNet. The standard deviation of $w^t - w^{t-K}$ for $K = 4, 5, 6$ at epoch 0 are 6.94×10^{-5}, 8.49×10^{-5}, 10^{-4}, at epoch 2 are 5.98×10^{-5}, 7.41×10^{-5}, 8.84×10^{-5}, at epoch 4 are 1.36×10^{-5}, 1.71×10^{-5}, 2.06×10^{-5}. Overall, a larger K leads to a more visible weights distribution shift whereas the variation in weights becomes less notable towards the end of training. With this observation, we set K different values depending on training phases. In general, a smaller K is used at the initial training stage where the weights drastically change and a relatively large K can be applied later when the model is close to convergence.

4.1.3 Grouped Parallelization. . From the view of the cloud, a fresh feature is used to perform both forward and backward steps, and the cloud weights are updated accordingly. From the view of the edge, the error gradients of batch t' are adopted to update w^t. We denote the iteration staleness at t as $s^t = t - t'$. In some cases, s^t could be unbounded. This would occur when there exists blocking in some queue. For example, if Q_1 blocks due to some network transmission issue, other queues would be quickly emptied. The edge would directly move to the next batch without waiting for feedback from the cloud. Ideally, full parallelization should allow s^t to be very large.

However, large s^t incurs problems in practice. First, the assumption of $w^t \approx w^{t'}$ does not hold anymore, resulting in random gradient descent direction and failure of convergence. Second, we do not have infinitely large queues to store the history information between t' and t. In particular, memory is quite limited on edge devices. Therefore, we restrict the value of s^t to be upper bounded by the parallelism factor K, and let it serve as



(a) BI-LSTM, IMDB (b) ResNet50, Transfer from Imagenet to CIFAR10) (c) ResNet50, Transfer from Imagenet to Tiny ImageNet)

Fig. 6. Validation accuracies and total training time of different models for varied Ks. No limit means we set a very large value for K and the performance is close to full parallelization.

a tunable knob in the tradeoff between efficiency and resources consumption: while a large K enables greater parallelization, it would consume more edge resources. When the user takes the error gradient of batch t' from $Q4$, it checks if $t - t' > K$. If it is, the user discards the results; otherwise, it applies feature replay for an update.

We use Fig. 5 to better display the running time for different configurations. In normal backpropagation where $K = 0$, all error gradients are fresh. In full parallelization, each module runs back to back, and the running time of the entire pipeline depends on the longest-running module ($P2$ in this example). In grouped parallelization, we divide the pipelined iterations into execution groups by K. For example, if $K = 2$, we tolerate at most two rounds of staleness. The third round has to wait for the return value from Q_4 to begin another iteration of the update, *i.e.,* another execution group. Feature replay can only be applied within a group but not across different groups.

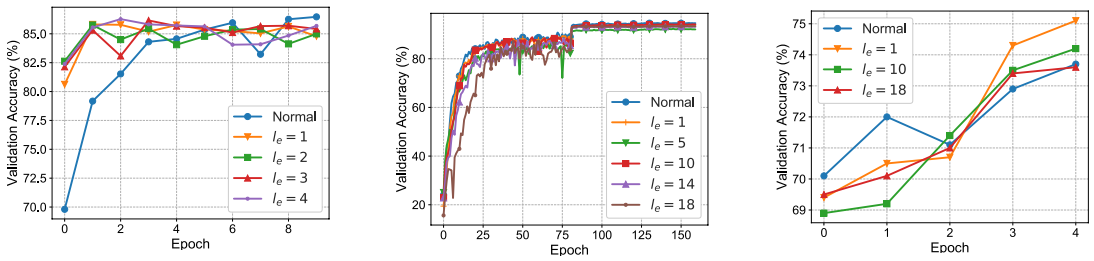
From the analysis, it is clear that the parallelism factor K plays a critical role in both accuracy and training latency. As observed from Fig. 4, the gap between w^t and $w^{t'}$ changes with the training progress and different models/datasets. That indicates that we should adopt different Ks depending on these factors. To better understand the relation between accuracy and training latency, we measure the two under a variety of Ks respectively for BI-LSTM and ResNet50. The results are shown in Fig. 6. Across different Ks for a single model, we can tell that while a larger K typically brings a lower latency, it leads to lower accuracy. Given that, one could set proper Ks to trade off latency and accuracy.

Observing Fig. 5(c)(d), we also found that different module proportions lead to varied running time, and hence it leaves design space to further speed up the parallel execution by choosing appropriate partition blocks, which we will discuss this in the following section.

4.2 Model Splitter

The model splitter divides the neural network at layer l_e. The input layer up to l_e is placed to run on the edge whereas the rest is deployed to the cloud. We only set one partition point for a model within one training iteration since the transmission cost would surpass the computational savings if we allow multiple transmissions in one iteration. After all, if we decide to offload features to the cloud, there is no motivation to send the error gradients back in a preemptive manner. Should l_e be changed across different training iterations, one shall consider the migration cost, *i.e.,* weights of the additional changed layers, between the edge and the cloud.

We perform tests to find out how accuracy would vary by l_es. In Fig. 7, we found that with different partition layer l_e, feature replay (with an appropriate K) achieves a similar convergence performance (with less than 1.5% accuracy loss), despite that different proportions of weights are updated by replayed features. Hence we think no matter how we change the partition block in the deployment, the convergence performance would not be significantly affected by feature replay.



(a) BI-LSTM, IMDB (b) ResNet50, CIFAR10, from scratch (c) ResNet50,Transfer ImageNet to Tiny ImageNet

Fig. 7. Validation accuracies with different l_es when training with the pipeline scheduler. We set: BI-LSTM on IMDB, $K = 4$ when epoch < 8 , $K = 5$ when epoch ≥ 8; ResNet50 on CIFAR10, $K = 5$; ResNet50 on Tiny ImageNet, $K = 5$ when epoch < 3, $K = 6$ when epoch ≥ 3.

4.3 Communication Optimizer

To reduce the transmission cost on the edge, we propose a communication optimizer to compress the features to be sent to the cloud (shown in Fig. 2 as a compressor). However, direct compression would introduce a significant loss of accuracy. The conventional error feedback SGD algorithm [23] cannot be applied to our case since features and error gradients, rather than the gradients, are transferred across the edge and cloud. Hence our design focus is a feature compression scheme with error compensation.

As we analyze, the feature quantization in the forwarding phase does not have much impact on the model performance, since the difference would be captured by the loss. It is the error gradient computed on the quantized features that cause an issue. With f denoting the loss function, the error gradient $\frac{\partial f(C(h_e))}{\partial C(h_e)}$ is biased from the original $\frac{\partial f(h_e)}{\partial h_e}$. Here we denote h_e as the output of the last layer on the edge l_e. Moreover, the quantization function $C(\cdot)$ is non-differentiable. Hence direct use of biased error gradients would result in convergence failure.

We propose to store the quantization error $E = C(h_e) - h_e$ on the edge, and use the error gradient $\hat{\delta}$ returned from the cloud, together with E to estimate the ground truth error gradient δ. We have the second-order Taylor expansion of f at $C(h_e)$:

$$\begin{aligned}\delta = \frac{\partial f(h_e)}{\partial h_e} &= \frac{\partial f(C(h_e) - E)}{\partial (C(h_e) - E)} \\ &\approx \frac{\partial f(C(h_e))}{\partial C(h_e)} - E \cdot \left(\frac{\partial^2 f(C(h_e))}{\partial C(h_e)^2}\right)^\top + o(E^2)I^2 \approx \hat{\delta} - E \cdot \left(\frac{\partial^2 f(C(h_e))}{\partial C(h_e)^2}\right)^\top .\end{aligned} \tag{7}$$

$\frac{\partial^2 f(C(h_e))}{\partial C(h_e)^2}$ is the Hessian matrix of $f(\cdot)$. It has been proved in [11, 56] that the outer product of the gradients is an asymptotically unbiased estimator of the Hessian matrix. Similarly, we use

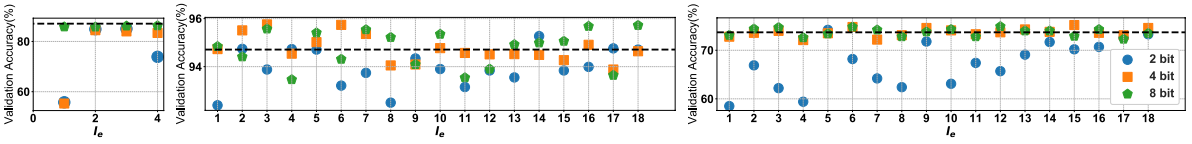
$$\boldsymbol{H}(C(h_e)) = \left(\frac{\partial f(C(h_e))}{\partial C(h_e)}\right)\left(\frac{\partial f(C(h_e))}{\partial C(h_e)}\right)^\top \tag{8}$$

to estimate the Hessian matrix. Hence the feedback can be completed efficiently with matrix multiplication. As the model weights, w gradually converge to the optimal weights w_*, feature h will gradually converge to the optimal h_* through w_*. Hence the cloud only needs to send $\hat{\delta}$ back, and the edge can calculate the Hessian matrix locally. Error Feedback Feature Compression is shown in Alg. 1.

Algorithm 1 Error-Feedback Feature Compression (on the edge, one iteration)

Input: batch of data x.

1: $h_e = F(x, w)$;
2: $E = C(h_e) - h_e$;
3: upload $C(h_e)$ to the cloud;
4: retrieve $\hat{\delta}$ from the cloud;
5: $\delta = \hat{\delta} - E \odot \hat{\delta} \odot \hat{\delta}$;
6: use δ to perform local backward step;



(a) BI-LSTM, IMDB (b) ResNet50 (Transfer Imagenet to CIFAR10) (c) ResNet50 (Transfer Imagenet to Tiny ImageNet)

Fig. 8. Error feedback compression with different bitwidths for l_e. For ResNet50 on CIFAR10 in transfer learning, we set $K = 5$ throughout the training process. In transferring to Tiny ImageNet, $K = 5$ when epoch < 3 and $K = 6$ when epoch ≥ 3. For BI-SLTM on IMDB, $K = 4$ when epoch < 8 and $K = 5$ when epoch ≥ 8. Dashed Lines represent normal backpropagation with full bitwidth (32 bit).

Function C can be any compression method. It is observed that, if we limit the accuracy drop to be within a certain threshold, the bit width of each layer could vary, enabling the possibility to explore the trade-off between communication overhead and accuracy performance. This factor is critical to bandwidth-stringent scenarios like IoT. As to the communication overhead, for example, an 8-bit quantization scheme can save the uploading cost by 4× with a slight increase in computational cost to calculate the true error gradients. For accuracy, we set l_e to each layer and vary its bit width to record the corresponding accuracy. Representative results are shown in Fig. 8. The dashed lines represent the accuracy results if we train models with normal backpropagation and full bitwidth. Such profiles provide the minimum bit width per layer given a maximum accuracy drop.

un

4.4 Profiler and Cost Model

From the previous statement, the parallelism factor K, per-layer bit width b have an impact on the final accuracy of the model and thus need to b profiled offline to estimate the online performance. However, factors such as feature size and running time on the edge are fundamental to the training latency, which should also be profiled offline. For a more fluent narrative, we refer readers to Sec. 5.2 for the profiling detail, while focusing on how the profiling results serve the cost model in this section.

To express the latency cost, we use T_1, T_2, T_3, T_4 to respectively denote the execution time of running P_1, P_2, P_3, P_4 with deployment configuration N. T_1 is composed of periods of the following stages: error feedback, feature replay, edge backward step, edge forward step, and feature compression. T_3 denotes the running time that the cloud performs forward and backward propagation. T_1, T_3 vary according to the partition point l_e, so we profile the execution time per model in advance. For the transmission latency T_2 and T_4, we use a linear model to estimate the uploading and downloading latency varying with feature or error gradient size:

$$T_2 = \frac{Size(h_e)}{r_u} \quad \text{and} \quad T_4 = \frac{Size(\delta)}{r_d}, \tag{9}$$

where r_u and r_d represent the uploading and downloading network speed. As the downloading speed is typically faster than the uploading speed, we only apply compression to the upload link and consider varied bit width per layer in $Size(h_e)$.

Combining the four modules, we let $T_M = \max\{T_1, T_2, T_3, T_4\}$ and $T_\Sigma = T_1 + T_2 + T_3 + T_4$ in the expression of total time cost as follows.

Cost of Normal Backpropagation: The cost model of τ-iteration normal backpropagation in the edge-cloud setting is quite simple since all operations are serialized:

$$\mathcal{M} = \tau \cdot T_\Sigma. \tag{10}$$

Cost of Full Parallelization: The cost model is similar to the model of the CPU pipeline. The only bottleneck lies in the hardware, *e.g.,* the rate limit of network transfer constrains the amount feature sent at each time. The module with the longest execution time becomes the parallelization bottleneck. Hence the cost model is:

$$\mathcal{M} = (\tau - 1) \cdot T_M + T_\Sigma. \tag{11}$$

In this model, the cost can be largely saved since most of the time, T_M is much smaller than T_Σ. In the ideal circumstance where four modules share the same execution time, $T_M = 1/4T_\Sigma$.

Cost of Grouped Parallelization: We first introduce the cost model of static grouped parallelization. In this model, K constrains the staleness gap between h_e^t and $\delta^{t'}$, and it divides τ iterations into $\lceil \frac{\tau}{K+1} \rceil$ groups. In each group, the grouped cost resembles the model in full parallelization. But at the end of each group, we take into account the serialization cost brought by serialized execution between groups.

Notice the calculation of the cost is not simply adding up the serialization parts T_Σ in the full parallelization cost model. Take the pipeline in Fig. 5 as an example. In deployment 1, T_2 is the bottleneck, and the group parallelization is as efficient as the full parallelization. While in deployment 2, the execution time of each part is relatively close, and P_1 of group 2 needs to wait for P_4 of group 1 to finish. Compared to full parallelization, the group parallelization increases the latency by $\max\{0, T_\Sigma - (K+1) \cdot T_M\}$. The cost model of static grouped parallelization is as follows:

$$\mathcal{M} = (\tau - 1) \cdot T_M + T_\Sigma + \left(\left\lceil \frac{\tau}{K+1} \right\rceil - 1 \right) \cdot \max\{0, T_\Sigma - (K+1) \cdot T_M\}. \tag{12}$$

This model can also describe the normal propagation model and the full parallelization cost model. If we set the K in Eq. 12 to 0, it is equal to Eq. 10. If we set the K in Eq. 12 to ∞, it is equal to Eq. 11. So two former circumstances can be viewed as special cases of static group parallelization.

In the dynamic deployment, we also need to consider the migration cost U in transferring the model weights between the edge and cloud. By assuming the final layer on the edge in the previous iteration is $l_e = l'$ and the current is l, we have

$$U(l', l) = \begin{cases} \dfrac{\Sigma_{i=l+1}^{l'} Size(L_i)}{r_u}, & l < l', \\ 0, & l = l', \\ \dfrac{\Sigma_{i=l'+1}^{l} Size(L_i)}{r_d}, & l > l'. \end{cases} \tag{13}$$

L_i denotes the sliced model layers. Therefore, the cost model of **dynamic grouped parallelization** is $\mathcal{M}(l_e, K, b, r_u, r_d, \tau) + U(l'_e, l_e)$.

To estimate the gap τ until the next triggering event, we use exponentially weighted moving average [20] as shown by Eq. 14. This is because τ is highly related to the data rate and bandwidth, whereas EWMA is a common method to estimate the dynamically changing bandwidth. Each time the decision engine is triggered, we update

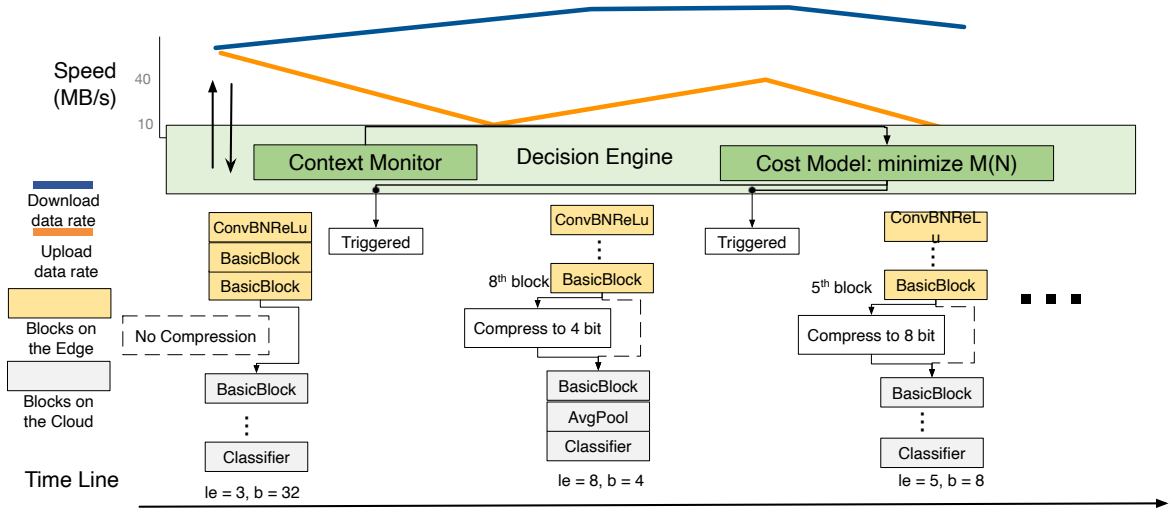


Fig. 9. An example of dynamic deployment on Resnet18 with 11 blocks. $K = 5$ in this case.

τ by

$$\tau^{t_m} = \beta \cdot \tau^{t_{m-1}} + (1 - \beta) \cdot (t_m - t_{m-1}), \tag{14}$$

where $t_m - t_{m-1}$ denotes the number of iterations between two consecutive engine trigger events.

4.5 Dynamic Decision Engine

So far we have introduced the edge-cloud training pipeline and two key methods resolving the parallelization and communication issues. As we have shown, the flexibility of the partition point, parallelism factor, and per-layer bit width embraces more possibilities for performance improvement. On the other hand, in the real world, the network condition is highly dynamic, especially where people are extremely mobile carrying the wearable/mobile devices around. In this section, we illustrate our dynamic decision engine which adaptively adjusts the deployment of the model according to the context.

Our decision engine constantly monitors the data rate of the uploading and downloading link, which are r_u, r_d respectively, and estimates the training latency with a cost model $\mathcal{M}$. When the engine detects the current deployment is suboptimal, it will trigger an event to alter the model across the edge and cloud. Each time the engine is triggered, we record the timestamp t and estimate the total number of iterations τ until the next time that the engine is triggered. The objective is to seek a deployment $N = (l_e, K, b) \in \mathcal{N}$ that

$$\underset{N \in \mathcal{N}}{minimize}\ \mathcal{M}(N, r_u, r_d, \tau) + U, \tag{15}$$

where U represents the migration cost from the previous deployment to N.

Search space $\mathcal{N}$ consists of three dimensions: l_e, K, b. For the choice of partition l_e, we only allow layer-wise cut after the activation layer or batch normalization layer following the principle of [1] that the output of ReLU potentially brings a higher compression ratio. As large volumes of data may be generated on the edge, it is inappropriate to handle those data purely on the edge and thus we exclude an edge-only training decision. K is profiled for each model on the calibration set that caps the staleness in the training loop and varies at different training stages. b denotes the per-layer bit width which determines the communication cost in combination with

Algorithm 2 Dynamic Decision Engine

Input: r_u, r_d, τ, current deployment $N^{t_m} = (l_e^{t_m}, K^{t_m}, b^{t_m})$.
Output: N^{t_m+1}.

$N^{t_m+1} = N^{t_m}$, estimated latency $\tilde{t} \leftarrow \infty$, temporary deployment N, current K^{t_m+1}.
for blocks on the edge $l_e \leftarrow 1 \cdots L$ **do**
  for $b \in \{2, 4, \ldots, 2^{|b|}\}$ **do**
    if $\mathcal{M}(l_e, K^{t_m+1}, b, r_u, r_d, \tau) + U(l_e^{t_m}, l_e) < \tilde{t}$ **then**
      $\tilde{t} \leftarrow \mathcal{M}(l_e, K^{t_m+1}, b, r_u, r_d, \tau) + U(l_e^{t_m}, l_e)$;
      $N = (l_e, K^{t_m+1}, b)$;
    end if
  end for
end for
if $N \neq N^{t_m}$ **then**
  Stop the edge from fetching new data;
  Wait until the history queue is empty;
  Deploy according to N;
  $N^{t_m+1} \leftarrow N$;
end if
return N^{t_m+1}

l_e. At each triggered event, we use a greedy algorithm to search for N which optimizes the cost, given r_u, r_d, and estimated τ.

Fig. 9 shows a toy example that how the deployment strategy varies across time on a model. The strategy alters by the monitored data rate. If the strategy is different from the previous one, the engine would re-deploy the model. When the deployment strategy changes from (3, 32) to (8, 4), the edge needs to retrieve model weights of the 4-th to the 8-th block from the cloud. Likewise, when the deployment changes from (8, 4) to (5, 8), the cloud would request the model weights of the 5-th to the 8-th block from the edge.

To sum up, our decision engine is given in Alg. 2. From the view of the edge, it feeds a batch of training data to the forwarding network and puts the feature into Q_1. Then the edge checks Q_4. If there exist error gradients in Q_4, the edge updates local weights by Eq. 3. From the edge transmission perspective, it repeatedly uploads elements from Q_2 to Q_3. From the view of the cloud, the cloud performs forward and backward steps on features in Q_2, and pushes the error gradients into Q_3. From the cloud transmission aspect, the cloud continuously pushes elements from Q_3 to Q_4. The decision engine, once triggered, interrupts the pipeline to re-deploy the model, and then the training pipeline resumes with updated deployment. The overall computational complexity of the algorithm is $O(L \cdot |b|)$, which is relatively efficient.

5 EVALUATION

In this section, we introduce the setup of our experiments and the implementation detail of the training pipeline. By both emulation and field tests in a variety of settings, we show how our proposed algorithms improve over the state-of-the-art.

5.1 Setup

5.1.1 Datasets and Models. To verify the effectiveness of our proposal, we select a set of representative machine learning tasks to deploy across the edge and the cloud in three typical application settings: split learning, transfer

Table 1. Datasets and Models

Dataset	Contents	Models	Train No.	Val. No.	Categories
CIFAR10	Images Classification	ResNet50, VGG19 [43] MobileNetV3(large) [24]	50000	10000	10
TinyImagenet (Sampled)	Images Classification	ResNet50 [16]	10000	1000	20
MovieLens	Click-Through Rate Prediction	DIN [57]	10^6	2×10^5	21
IMDB	Textual reviews	BI-LSTM [13]	50000	50000	2
3DChair	Images Generation	Chair [9]	45000	15000	1393

learning, and cross-silo federated learning. All datasets and models are provided in Table 1. Now, we introduce each task by settings as follows.

Split learning performs training from the scratch over a model residing at both the edge and the cloud. Both discriminative and generative models are included, and three datasets are chosen: CIFAR10 [25], 3DChair [2] and IMDB [32]. Among them, 3Dchair is a dataset containing 1393 aligned chair models, each rendered from 62 viewpoints. The input is an 825-long vector and the output is generated images of 128×128 pixels.

Compared to split learning, **transfer learning** fine-tunes pre-trained models to generate personalized models, which is more common in edge computing. We test two types of transfer learning — fine-tuning the entire pre-trained model (marked as Type 1) and freezing parts of the model and fine-tuning the rest (as Type 2). We download a pre-trained ResNet50 over ImageNet [8] from `torchvision` package. For Type 1, the pre-trained model is transferred to TinyImageNet (sampled) [28] and CIFAR10 respectively. TinyImagenet is from the same source domain of ImageNet, aiming to explore adaptation efficacy of our scheme. Since CIFAR10 is from a different source domain, it is used to show transferability. For Type 2, the model is re-trained with the first 10 blocks frozen. We argue that the datasets are close to that in real-world applications where CIFAR10 (32×32) represents a small input size and TinyImageNet (224×224) stands for a large input size.

Cross-silo federated learning is tested for the Click-Through Rate (CTR) Prediction task on MovieLens. Deep Interest Network is used as our backbone model. In this setting, we assume two independent devices each keep half of the movie features while information of advertisements is known to all. The top MLP model is at the cloud predicting users' CTR. The training procedure follows the conventional federated learning setting except that each device interacts with the cloud according to our proposed training pipeline.

All models chosen are state-of-the-art and have great potential for being deployed in real-world application scenarios. Their training hyperparameters, shown in Table 2, are tuned by normal training to attain the best performance of each model. Without particular explanation, all experimental settings default to Table 2. Among these models, ResNet50, VGG19, MobileNetV3, DIN, and BI-LSTM are *discriminative* CNN or RNN models modeling $P(y|x)$. x is the input and is generated from users' raw data. We need to protect the privacy of raw data, and thus x is always kept local. Layers close to input shall be on the edge and the rest are on the cloud.

Chair is a *generative* model for modeling $P(x, y)$. Different from the discriminative models, g_θ of Chair is used to generate x from a low-dimensional vector $z \sim \mathcal{N}(0, I)$: $x = g_\theta(z) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. As a result, the noise z is unimportant, but the generated x could leak user privacy. In [15], it is pointed out that the data used to train generative models could be sensitive, for example, human faces which can be used to identify an individual. If we let x be generated on the cloud, it would leak users' identities. Hence we keep x local in the deployment of generative models. The structure is reverse to that of the discriminative model, and thereby poses a very different challenge: the cloud generates random noise and forwards it through the neural network. The edge takes features from the cloud to update its model to minimize the discrepancy between the generated output and users' ground truth.

Table 2. Hyperparameter setting for training different models

	ResNet50 (CIFAR10)	Mobile-netv3 (large)	VGG19	Chair	BI-LSTM	ResNet50 (Tiny-Imagenet)	ResNet50 (Imagenet to CIFAR10)	DIN
Data Augmentation	✔			✗	✔			✗
Optimizer	SGD			Adam			RMSprop	SGD
Initial Learning Rate	0.1			5×10^{-4}	10^{-3}	8×10^{-5}	5×10^{-5}	0.1
Momentum	0.9			–				
Weight Decay	10^{-4}			–		0.001	–	
Epochs	160	200	100	10		5		
Scheduler	Multi-Step			–		Multi-Step	–	Exp
Scheduler Decay Factor	0.1			–				0.1
Step Epoch	80,120	100,150,180	80	–		3	–	

Table 3. The size of intermediate features of different models with batch size 32 in MB.

ResNet50		ResNet50 TinyImg		Mobilenetv3(large)		VGG19		DIN		Chair		BI-LSTM	
l_e	Size	l_e	Size	l_e	Size	l_e	Size	l_e	Size	l_e	Size	l_e	Size
1	8.00	1	24.50	1–2	2.00	1	8.00	1	0.125	1	0.05	1	3.13
2–4	32.00	2–4	98.00	3–4	3.00	2	2.00	VGG19 (cont)		2–3	0.03	2	6.25
5–8	16.00	5–8	49.00	5–7	1.25	3	4.00	12–15	0.25	4	0.50	3	3.13
9–14	8.00	9–14	24.50	8–11	0.63	4	1.00	16	0.06	5–6	2.00	4	0.02
15–17	4.00	15–17	12.25	12–13	0.88	5–7	2.00	17–18	0.50	7–8	2.88		
18	0.25	18	0.25	14–16	0.31	8	0.50			9–10	6.00		
				17	0.16	9–11	1.00						

5.1.2 Edge Devices and Cloud Servers. GPU server with NVIDIA RTX 2080Ti with 11GB memory serves as our cloud. We select two types of edge devices: *NVIDIA JetsonTX2* is a popular AI edge computing and IoT device. It is equipped with NVIDIA Pascal architecture with 256 CUDA cores, Dual-core NVIDIA Denver 2 64-bit CPU, quad-core Arm Cortex-A57 MPCore processor with ARM64 OS, and a shared 8 GB 128-bit LPDDR4 shared memory between GPU and CPU. It works under a maximum of 15W power consumption. *Raspberry Pi 4B* is a portable edge device with Quad-core Cortex-A72 (ARM v8) 64-bit CPU and 4GB LPDDR4-2400 memory with aarch64 OS. Both devices have WiFi modules an Ethernet port. We implement our algorithm in `Pytorch` and the transmission protocols follow the guideline of socket programming.

5.2 Profiling

As aforementioned, offline profiling is required to obtain some characteristics of the model, which are needed in the cost calculation in the online phase. We introduce the profiling details in this section.

Profiling feature/model size: the feature size decides the transmission latency and varies per model. We show the profiling results in Table. 3. The size of ResNet and VGG are comparatively larger than MobileNet as the MobileNet is specifically designed for mobile devices. The feature size also depends on the input size: for large inputs on Tiny ImageNet, the feature size is comparatively larger, posing as a non-trivial factor to the latency

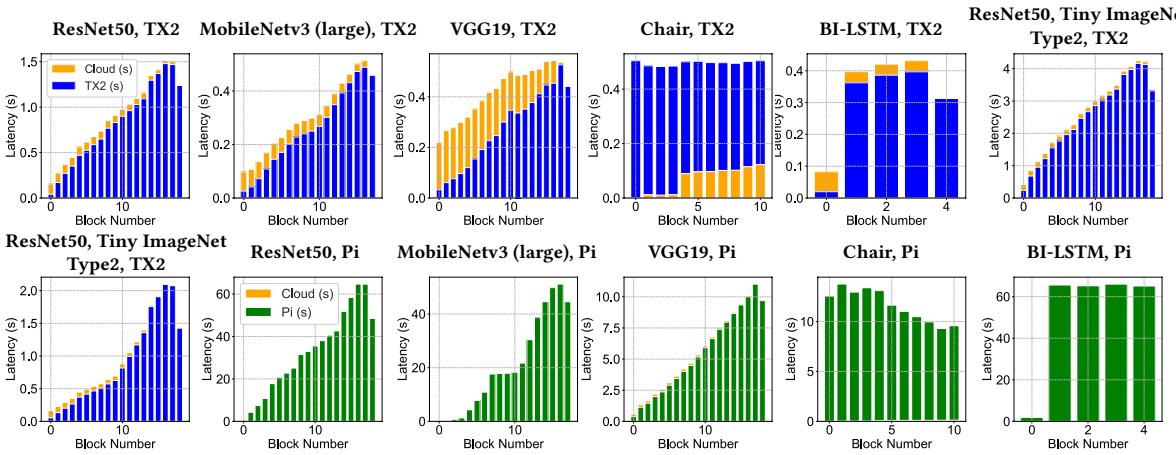


Fig. 10. Running time on different devices for various models. X-axis represents the block indices of the partition point. Latency is the average latency per iteration running datasets with batch size 32.

cost. The profiling of model size is very similar to that of feature size. The only difference is to calculate the size of the entire weights tensor rather than the last-layer output of each block.

Profiling running time: As pointed out by [45], it is inefficient to use computational operations to estimate the running time. We take the characteristics of each objective platform into account. To precisely profile T_1 and T_3, we run the model on actual devices. Due to the inner working mechanism of GPUs, it is inaccurate to profile a single block's running time. Hence we choose to profile the duration that the model runs up to that point. For example, if the first 3 blocks are deployed to the edge, we measure the running time of forwarding consecutive 3 blocks and its corresponding backward step on the edge. If the edge cannot run the full block due to memory limitation, we do not profile the result on the device.

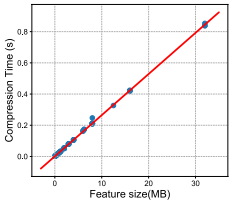
From Fig. 10, we can tell that cloud processing is much faster than TX2 and its running time is negligible compared to that on Pi. The overall running time increases with more blocks deployed on the edge, but the specific latency varies by device and model. On models with Skip-connections, like ResNet and MobileNet, the cloud is even faster. For Chair, the overall running time is close despite different proportions on edge and cloud. As a result, the network is a critical factor and our system would choose the deployment with minimal transmission cost.

We also profile the compression time on edge devices. On devices, the compression time cannot be omitted due to the limited CPU power. We quantize features of different sizes to 8-bit representation and record the compression overhead on TX2 and Raspberry Pi in Fig. 11. The compression time is linear to the feature size, so we apply the corresponding linear models in estimating T_1. As decompression is conducted on the cloud, so we neglect its latency cost.

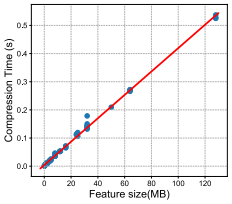
5.3 Convergence and Accuracy

Before implementing our real-world system, we first show the simulation results on servers to verify the convergence and accuracy of our proposed methods.

5.3.1 Pipeline Training with Feature Replay. Due to the limited computational resource and unstable connectivity of edge devices, large-batch training is constantly not available on the edge. Hence we would like to figure out how batch size affects the feature replay. In Fig. 12, we can see for small or large batch sizes, feature replay can



(a) Comp. Time on Pi



(b) Comp. Time on TX2

Fig. 11. The profiled time of compressing features to 8-bit representations on edge devices. It follows a linear model on TX2: $y = 0.00415 \times x + 0.0041$ and Pi: $y = 0.02636 \times x + 0.00114$.

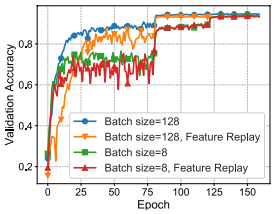


Fig. 12. The convergence curve of Resnet50 on CIFAR10 with and without group parallelization ($K = 5$, $l_e = 18$) under different batch sizes.

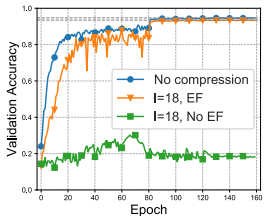


Fig. 13. The convergence curve of Resnet50 on CIFAR10, respectively when the model is without any compression, with quantization, and with error-feedback compression.

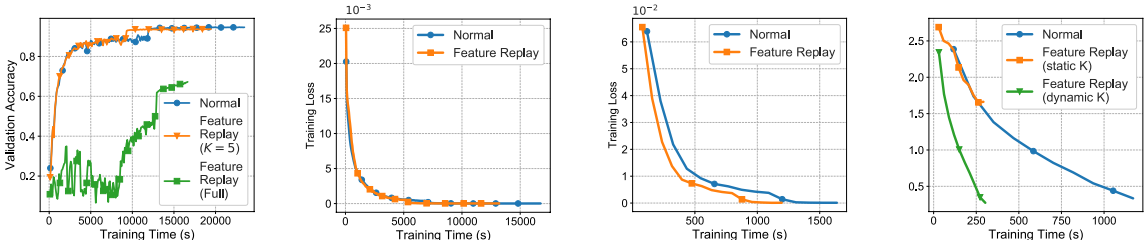
achieve almost the same convergence and accuracy as that without applying feature replay. Hence we consider batch size does not have a significant impact on our method. Without particular mentioning, the default batch size is set to 32 in this paper.

To find out how the pipeline training converges compared to normal backpropagation, we compare the training curves on models in Fig. 14 and the final accuracies in the first two columns of Table 4. Although the validation accuracy quickly ramps up at the beginning of full parallelization, it fails to converge in the end. The grouped parallelization ($K = 5$) achieves almost the same accuracy as the normal backpropagation but with less wall clock time. On different CNN models, using pipeline training will not affect the convergence, but provide significant speedup. The accuracy loss can be kept under 1% for most cases. Especially, when we perform feature replay in transfer learning, the accuracy loss is almost negligible since fine-tuning on pre-trained models causes less weights variation despite that the models are pretrained on the same source domain (ImageNet to TinyImageNet) or a different domain (ImageNet to CIFAR10). On bidirectional LSTM, since the backpropagation chain in LSTM cells is much longer than residual blocks, there are some accuracy losses, which can be largely mitigated by our dynamic adjustment of K. Overall, our pipeline training with feature replay converges faster than normal training with negligible accuracy loss. However, since the transmission of features is very fast in the simulation, our algorithm cannot fully benefit from pipeline training. We will further show the advantage in the next section.

5.3.2 Compression with Error Feedback. We compare the convergence performance when the model is without any compression method, with quantization only, and with error-feedback quantization in Fig. 13. We particularly choose $l_e = 18$ so that error feedback has the most severe impact on the model as it is close to the rear of ResNet50. But there is no clear difference in the final accuracy between the one with error feedback and that without compression. The one without error feedback but with quantization fails to converge. We also show the final accuracy of training with 8-bit compression and error feedback in the third column of Table 4. For most of the models, the accuracy loss can be kept under 1% demonstrating our communication optimizer has little impact on accuracy.

6 COMPARISON WITH BASELINES

In this section, we implement our system in both emulation and field tests under a variety of settings and discuss the experimental results in comparison with baselines.



(a) ResNet50, CIFAR10, Transfer Type1 (b) MobileNetV3(large), CIFAR10 (c) ResNet50, Tiny Img, Transfer Type1 (d) BI-LSTM, IMDB

Fig. 14. Comparison of convergence performance between normal back-propagation and pipeline training on different models. For ResNet50 and MobileNetV3 on CIFAR10, we set $K = 5$ throughout the training process. For ResNet50 in transfer learning, $K = 5$ when epoch < 3 and $K = 6$ when epoch ≥ 3. For BI-SLTM on IMDB, $K = 4$ when epoch < 8 and $K = 5$ when epoch ≥ 8.

Table 4. The validation accuracies (%, or loss for Chair) of models with pipeline scheduler and communication optimizer (8-bit compression with error feedback). Type 1 and 2 refer to transfer learning and the rest perform split learning.

Metrics	Models, Datasets	Normal	Pipeline Training	Pipeline Training with EF
Accuracy	ResNet50, CIFAR10	94.71	93.48	95.16
	MobileNetV3(large), CIFAR10	91.51	92.49	92.06
	VGG19, CIFAR10	91.57	91.06	91.82
	BI-LSTM, IMDB	87.08	85.62	86.29
	Type 1, CIFAR10	93.80	93.39	94.24
	Type1, Tiny Img	73.7	74.6	73.6
	Type2 Tiny Img	70.7	70.2	70.9
	DIN, MovieLens	73.17	73.09	73.09
Loss	Chair, 3DChair	$4.7e^{-4}$	$4.9e^{-4}$	$5.2e^{-4}$

6.1 Baselines

We choose two typical methods as our baselines: *JointDNN* [10] and edge-only training. JointDNN is a representative work on the edge-cloud training, which finds out the optimal partition for a fixed DNN model under constant network conditions. It searches the shortest path on a resources DAG. The edge-only training is the default way for the edge to perform training on local data. Cloud-only training is out of the scope since direct upload of user data would be privacy-leaking.

6.2 Network Conditions

We evaluate our system over different network conditions. 4G: We use HEVC 4G/LTE traces [48] as the network traces. The traces measure 4G networks along several routes in and around the city of Ghent, Belgium, on a Huawei P8 Lite mobile phone. The traces are collected in different contexts including on bike, on bus, on foot, in a car, in a tram, or on train. We reformat the traces and use Mahimahi network emulation tool [33] to replay these 4G network traces. **WiFi** is the most ubiquitous mobile network and a typical WiFi bandwidth

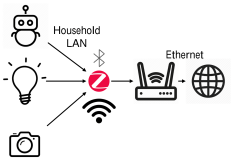
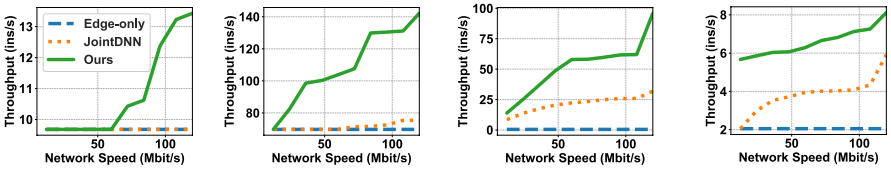


Fig. 15. The illustration of one-hop wireless connection to Ethernet.



(a) ResNet50 TinyImagenet on TX2 (b) MobileNetV3 (large) CIFAR10 on TX2 (c) BI-LSTM on Pi (d) Chair on Pi

Fig. 16. Achieved throughputs vs. network speed for various ⟨model, datase, devicet⟩ configurations. The bitwidth is 8.

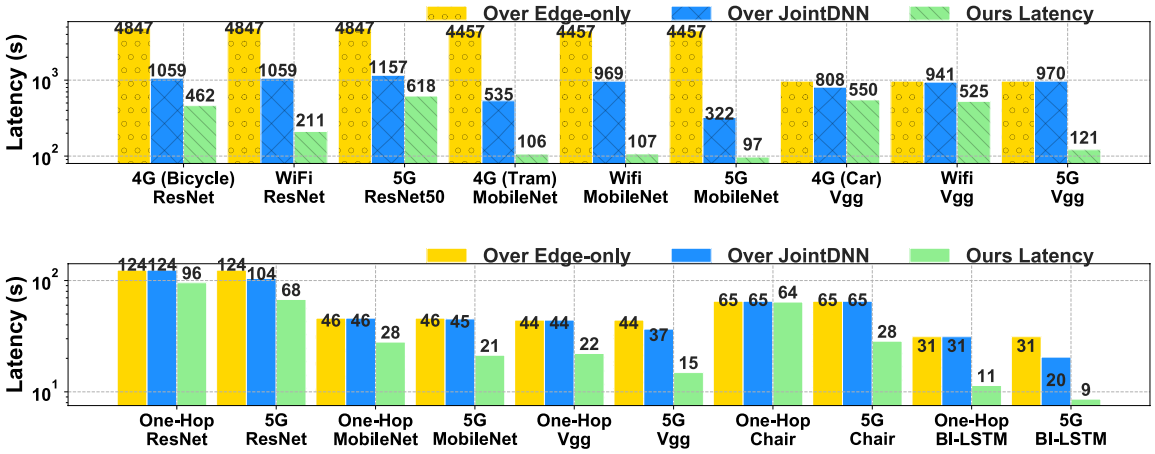


Fig. 17. Upper: Raspberry Pi. Lower: TX2. Latencies of different models over various network conditions. The default batch size is 32 and the latency has a unit of 'seconds/100 batches.'

condition is shown in Fig. 18. **One-hop wireless connection to Ethernet** is another common network setting in household IoT architectures shown in Fig. 15. People use IoT appliances and robots in home automation and these devices are connected to the Ethernet through a gateway. The one-hop wireless connection could be WiFi, BLE, Zig-bee, etc., and we use WiFi in the experiments. A period of sampled network conditions is provided in Fig. 18. **5G** symbolizes the next generation of mobile networks which has promising applications in IoT settings [35, 40]. We use 5G networks provided by our local ISP and establish a real cloud environment on Elastic Cloud Server provided by Huawei.

6.3 Throughput Analysis

We first compare our system's throughputs in controlled experiments where we fix the network speed by Mahimahi. According to Table 3, feature sizes are largest in ResNet50 for Tiny ImageNet and small in MobileNetV3, and thus the two models are chosen for TX2 experiments. Due to the memory limitation of Pi, we choose BI-LSTM and Chair as two representative models for large and small features respectively. We report the number of training instances processed per second at different speeds in Fig. 16. The throughput is the highest in our system across all network rates and tends to grow with the network speed. It is evidence that our pipeline training can better

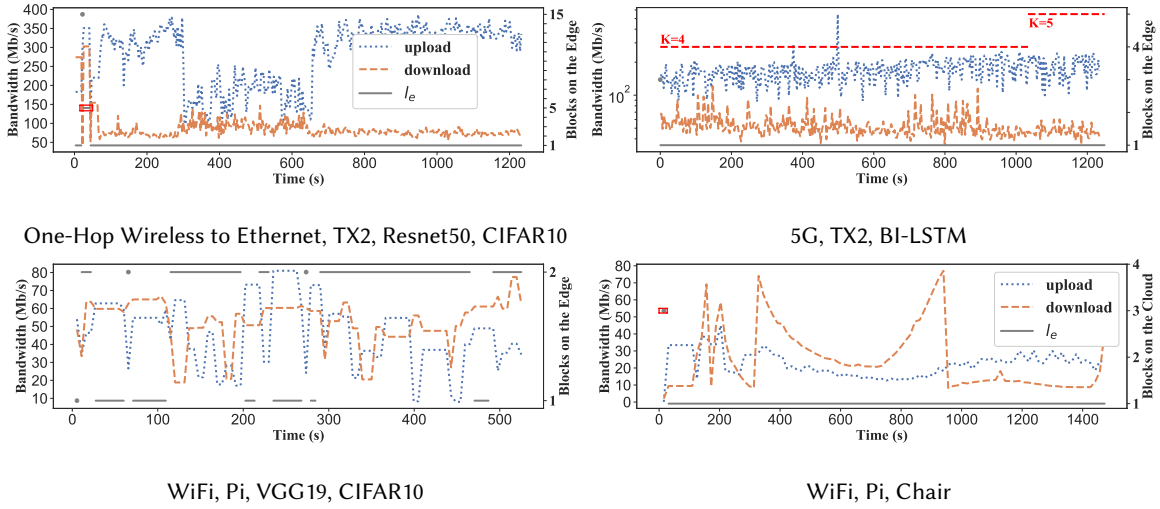


Fig. 18. The real-world upload and download link bandwidth, and the varying of l_e in the pipeline training. For Resnet50, VGG19 and Chair, K is kept default to 5 during training. For BI-LSTM, the decision engine adopts dynamic K policy.

take advantage of the high network speed while defaulting to the edge-only training when the bandwidth is poor. By the profiling results in Fig. 10, the total running time is close for different partitions on Chair, and thus there is less gain in modifying the configurations.

Under real-world network conditions, we measure the running time of training 100 batches with 32 instances per batch in Fig. 17. On Raspberry Pi, our system achieves the lowest latency in all contexts compared to baselines, saving up to 97.6% latency. It only takes several minutes to finish what takes more than an hour in the edge-only case for ResNet and MobileNet. TX2, on the other hand, is more powerful in managing the training load and gains less benefit when the network connectivity is poor. However, when the network speed is fast, our system still significantly reduces the training latency. For simple models such as BI-LSTM, our system attains an impressive latency of 9 seconds for processing 100 batches, feasible in simultaneous interpreting scenarios.

The flexibility of our system lies in the ability to take advantage of the context while avoiding rash decisions. To examine more closely, we choose some representative cases to analyze in Fig. 18. The batch size is set to 128 for TX2 for better illustration (more to transmit per iteration). We can see that, when the bandwidth has a sudden surge, our system can seize the opportunity to offload as much as possible to the cloud (One-Hop, TX2, Resnet50). When the bandwidth drops suddenly, maybe due to loss of connection, our system does not change deployment immediately (5G, TX2, BI-LSTM) since it is unwise to transmit model parameters under weak network conditions. But if the data rate continues to be low, our system will automatically seek better deployment (WiFi, Pi, VGG19). Our system fully exploits the computational power on the edge when the network condition is poor but uses the cloud to accelerate training when the connectivity is good. On Chair, as different deployments lead to similar running time, the bottleneck lies in the transmission. Hence our system chooses a configuration with minimal communication overhead (Wifi, Pi, Chair).

6.4 Accuracy and Latency

For better comparison with baselines, we record the validation accuracy varying with the training time until convergence in each setting.

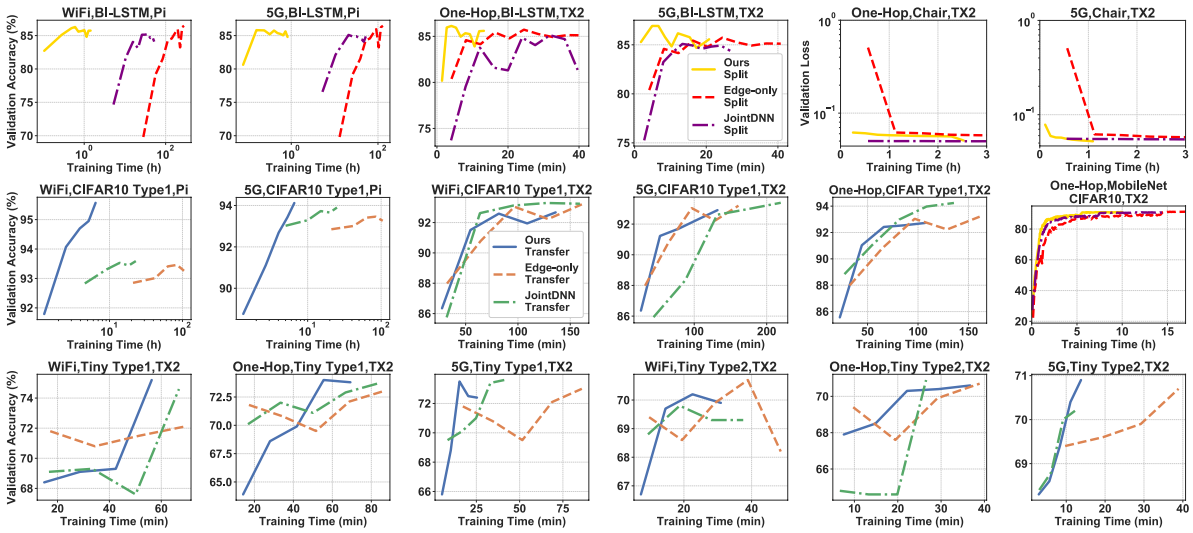


Fig. 19. Convergence with training batch 32 in a variety of settings. The first row and MobileNetv3 are results of split learning (training from scratch). The rest are results of transfer learning.

Split learning is performed on Chair, BI-LSTM, and MobileNetv3 with results in Fig. 19. Our method is superior to JointDNN and edge-only training, particularly on Pi. A base 10 log scale is used for the X-axis of results on Pi since it would take hours for baselines to converge. Our pipeline training typically converges within an hour with an exception on MobileNetv3 training from scratch for CIFAR10. But still, it only takes almost half of the time compared to baselines. The training time is acceptable in the case where high-quality images such as medical images, are collected at a low rate (in a patient's body) while being simultaneously trained.

In **transfer learning**, we transfer ResNet50 pre-trained on ImageNet to Tiny ImageNet on TX2 and to CIFAR10 on TX2 and Pi. The convergence curves are given in Fig. 19. Under the fast but highly fluctuated 5G network, our system benefits most from the wireless conditions to gain significant speedup. For Type 1 transfer, our system gets comparable train-from-scratch accuracy by fine-tuning the entire model within tens of minutes. Type 2 transfer further gains speedup by sacrificing some accuracy. In Type 2 transfer, the error gradients are not sent back to the edge, resembling the inference task. Hence we believe our system has low latency in inference tasks as well.

In **cross-silo federated learning (FL)**, we have either two TX2 devices or one TX2 and one Pi as clients contributing features to the server. The server fetches one uploaded feature from Q_2 of each client and synchronizes them to ensure they belong to the same identity. We mainly compare the performance against the conventional cross-silo FL and report the accuracies computed on the server in Fig. 20. In heterogeneous networks and devices, our system reduces the overall training latency almost without accuracy loss compared to conventional FL, with exceptions for (TX2 5G, Pi 5G) and (TX2 WiFi, Pi 5G). We will further analyze the two cases.

Different from standalone training, FL can be dragged behind by the varied computational power of clients. Under hybrid modes, TX2 computes extremely fast and has to wait for synchronization with Pi on the server. As we depict the actual staleness $t - t'$ on each device in Fig. 21, we can tell that $t - t'$ equals 5 constantly on TX2 (we set $K = 5$) while equals to 0 most of the time on Pi. Hence the network condition is never a bottleneck for Pi and the system's performance is close to the standard FL. Also, from Fig. 21, we can see that when the settings are symmetric, such as (TX2 WiFi, TX2 WiFi), the actual staleness distributes uniformly representing an

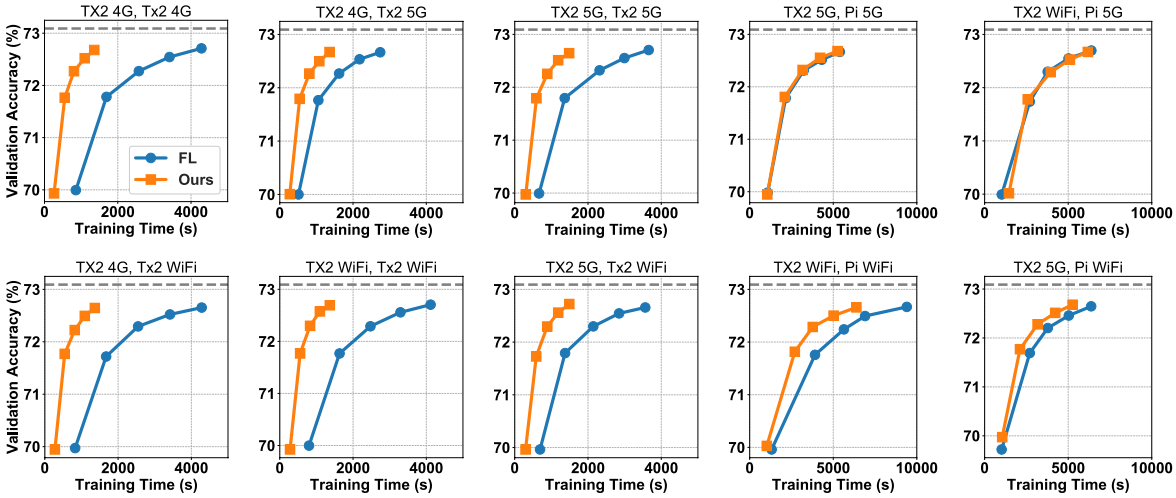


Fig. 20. Convergence of cross-silo FL in a variety of settings. Dashed lines represent the best accruracies in standalone training mode.

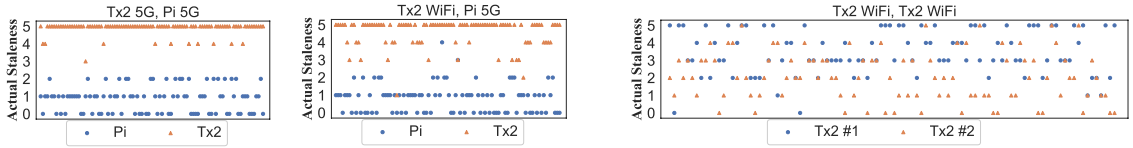


Fig. 21. The acutal staleness $t - t'$ in cross-silo federated learning under different network conditions. We randomly pick 100 iterations to show.

effectively working pipeline, and the speedup is more notable. If we remove the synchronization barrier in FL, the efficiency of the system can be further improved.

6.5 Memory Cost

On IoT, mobile, or other edge devices, the on-device resource is quite limited and thus requires special attention. We log the memory cost on devices (JetPack 4.5 OS and Raspbian OS) while our system is running, and record the results of repeated experiments. For cross-silo FL, the usage is averaged over two devices.

As shown by Fig. 22, since Raspberry Pi is more stringent on memory, our system is prone to offload more computation to the cloud. On average, our system requires merely 1% more memory than JointDNN and reduces the memory cost by 19.51% on average from edge-only. On TX2, our system consumes slightly more memory than JointDNN since we need to store data in the history queue for future feature replay. But since our system can balance the four modules for parallelism, the memory increase is minimal. Compared with the edge-only method, our system requires far less memory when the partition layer is away from the output. In the case where the partition choice is close to the output, our method requires on average 2.94% more memory than edge-only training. Hence we conclude our proposed method is resource-friendly for edge devices.

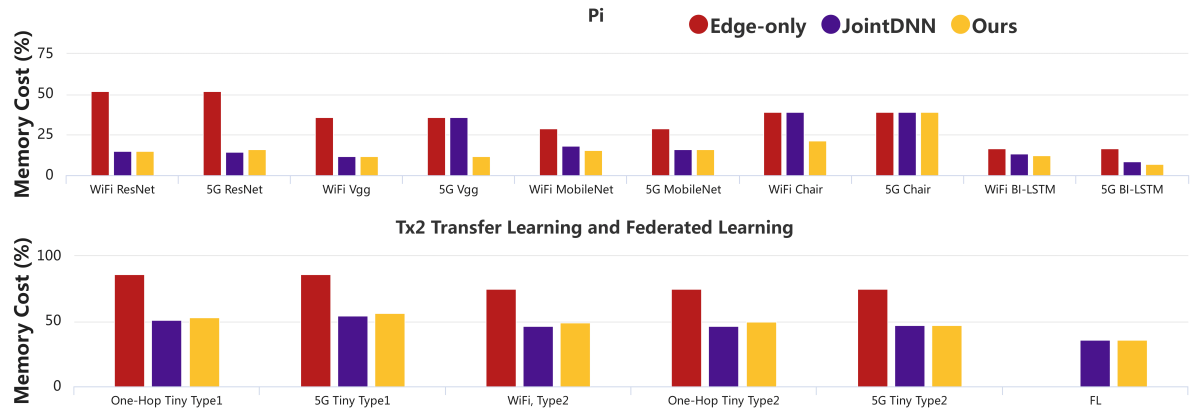


Fig. 22. The comparison of average memory costs between our methods and baselines in varied contexts.

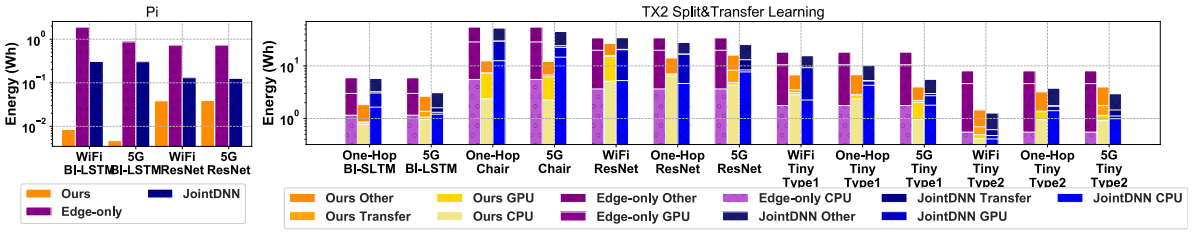


Fig. 23. The comparison of energy concumptions between our methods and baselines in varied contexts. Each color block represents the consumption of a particular component.

6.6 Energy Consumption

We also quantify the energy consumption in different deployment scenarios in comparison with baselines. For TX2 (MAX-N configuration), we use OS probes on Jetson to measure the energy consumption of four components: *Transfer* means the energy cost of sending or receiving data; *GPU* is for DNN computation; *CPU* is for compression, scheduling, and detaching data from GPU; *Other* represents the rest. For Pi, we add an FNIRSI-C1 Type-C-USE power meter between Pi and the power adaptor to monitor the overall energy consumption. Results are reported in Fig. 23.

On Pi, our energy consumption remains low which is suitable for edge devices. For TX2, our system costs much less energy in total than baselines. Our CPU energy is the leading cost, due to the compression and scheduling performed. But it is no worse than JointDNN which busy awaits for the error gradients to return when no actual task is running. Our system trades a part of the transfer cost for big savings in the computation energy, suitable for energy-scarce IoT devices.

7 CONCLUSION

We address the important issue of model personalization by proposing a novel and practical DNN training pipeline across the edge and cloud. We essentially rebuild the training framework customized for the edge-cloud computing scenario, taking into account the dynamic nature of the edge. Our system automatically selects a deployment strategy fitting to the context. Despite all the speed-up efforts, our method incurs almost no

accuracy loss. Emulation and field tests in a variety of settings show our system reduces training latencies significantly compared with baselines, yet without much burden on memory and energy costs. To sum up, our system contributes as a prototype for efficient on-device DNN (including CNNs and RNNs, discriminative and generative models) training.

ACKNOWLEDGMENTS

This work was partially supported by National Key Research and Development Program of China under Grant No. 2020YFB1708700, No. 2018AAA0101202, National Science Foundation of China under Grant No. 61902245, No. 62032020, No. 62136006, No. 61960206002, No. 42050105, No. 61829201, the Science and Technology Innovation Program of Shanghai under Grant No. 19YF1424500 and College Student Innovation Program of Shanghai Jiaotong University under Grant No. IPP23075. We thank our anonymous reviewers for their thoughtful comments and suggestions.

REFERENCES

[1] Mario Almeida, Stefanos Laskaridis, Stylianos I. Venieris, Ilias Leontiadis, and Nicholas D. Lane. 2021. DynO: Dynamic Onloading of Deep Neural Networks from Cloud to Device. arXiv:cs.DC/2104.09949

[2] Mathieu Aubry, Daniel Maturana, Alexei A Efros, Bryan C Russell, and Josef Sivic. 2014. Seeing 3d chairs: exemplar part-based 2d-3d alignment using a large dataset of cad models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3762–3769.

[3] Lukas Cavigelli, Georg Rutishauser, and Luca Benini. 2019. EBPC: Extended Bit-Plane Compression for Deep Neural Network Inference and Training Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 4 (2019), 723–734. https://doi.org/10.1109/JETCAS.2019.2950093

[4] Mingqing Chen, Rajiv Mathews, Tom Ouyang, and Françoise Beaufays. 2019. Federated learning of out-of-vocabulary words. *arXiv preprint arXiv:1903.10635* (2019).

[5] Hyomin Choi and Ivan V Bajić. 2018. Deep feature compression for collaborative object detection. In *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 3743–3747.

[6] Xiangfeng Dai, Irena Spasić, Bradley Meyer, Samuel Chapman, and Frederic Andres. 2019. Machine learning on mobile: An on-device inference app for skin cancer detection. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 301–305.

[7] Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, et al. 2012. Large scale distributed deep networks. (2012).

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[9] Alexey Dosovitskiy, Jost Tobias Springenberg, Maxim Tatarchenko, and Thomas Brox. 2016. Learning to generate chairs, tables and cars with convolutional networks. *IEEE transactions on pattern analysis and machine intelligence* 39, 4 (2016), 692–705.

[10] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. 2019. JointDNN: an efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing* (2019).

[11] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York.

[12] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. *Baidu Research, Tech. Rep.* (2017).

[13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 6645–6649.

[14] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 123–136.

[15] Jamie Hayes, Luca Melis, George Danezis, and ED Cristofaro. 2017. LOGAN: Evaluating Information Leakage of Generative Models Using Generative Adversarial Networks. *arXiv preprint arXiv:1705.07663* (2017).

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[17] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE.

[18] Jin Huang, Colin Samplawski, Deepak Ganesan, Benjamin Marlin, and Heesung Kwon. 2020. CLIO: enabling automatic compilation of deep learning pipelines across IoT and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and*

Networking. 1–12.
[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (2018).
[20] J Stuart Hunter. 1986. The exponentially weighted moving average. *Journal of quality technology* 18, 4 (1986), 203–210.
[21] Zhouyuan Huo, Bin Gu, and Heng Huang. 2018. Training neural networks using features replay. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 6660–6669.
[22] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 615–629.
[23] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*. PMLR, 3252–3261.
[24] Brett Koonce. 2021. MobileNetV3. In *Convolutional Neural Networks with Swift for Tensorflow*. Springer, 125–144.
[25] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[26] Matthias De Lange, Xu Jia, Sarah Parisot, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2020. Unsupervised Model Personalization While Preserving Privacy and Scalability: An Open Problem. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
[27] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. 2020. SPINN: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–15.
[28] Ya Le and Xuan Yang. 2015. Tiny imagenet visual recognition challenge. *CS 231N* 7, 7 (2015), 3.
[29] B. Lin, Y. Huang, J. Zhang, J. Hu, X. Chen, and J. Li. 2019. Cost-Driven Offloading for DNN-based Applications over Cloud, Edge and End Devices. *IEEE Transactions on Industrial Informatics* (2019).
[30] Ching-Yi Lin and Radu Marculescu. 2020. Model Personalization for Human Activity Recognition. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 1–7. https://doi.org/10.1109/PerComWorkshops48775.2020.9156229
[31] Fenglin Liu, Xian Wu, Shen Ge, Wei Fan, and Yuexian Zou. 2020. Federated learning for vision-and-language grounding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 11572–11579.
[32] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 142–150.
[33] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate record-and-replay for {HTTP}. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 417–429.
[34] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, et al. 2021. Federated Evaluation and Tuning for On-Device Personalization: System Design & Applications. *arXiv preprint arXiv:2102.08503* (2021).
[35] Quoc-Viet Pham, Fang Fang, Vu Nguyen Ha, Md. Jalil Piran, Mai Le, Long Bao Le, Won-Joo Hwang, and Zhiguo Ding. 2020. A Survey of Multi-Access Edge Computing in 5G and Beyond: Fundamentals, Technology Integration, and State-of-the-Art. *IEEE Access* 8 (2020), 116974–117017. https://doi.org/10.1109/ACCESS.2020.3001277
[36] Maarten G Poirot, Praneeth Vepakomma, Ken Chang, Jayashree Kalpathy-Cramer, Rajiv Gupta, and Ramesh Raskar. 2019. Split learning for collaborative deep learning in healthcare. *arXiv preprint arXiv:1912.12115* (2019).
[37] Daniele Romanini, Adam James Hall, Pavlos Papadopoulos, Tom Titcombe, Abbas Ismail, Tudor Cebere, Robert Sandmann, Robin Roehm, and Michael A. Hoeh. 2021. PyVertical: A Vertical Federated Learning Framework for Multi-headed SplitNN. arXiv:cs.LG/2104.00489
[38] Joe Saunders, Dag Sverre Syrdal, Kheng Lee Koay, Nathan Burke, and Kerstin Dautenhahn. 2016. "Teach Me - Show Me"–End-User Personalization of a Smart Home and Companion Robot. *IEEE Transactions on Human-Machine Systems* 46, 1 (2016), 27–40. https://doi.org/10.1109/THMS.2015.2445105
[39] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:cs.LG/1802.05799
[40] Kinza Shafique, Bilal A. Khawaja, Farah Sabir, Sameer Qazi, and Muhammad Mustaqim. 2020. Internet of Things (IoT) for Next-Generation Smart Systems: A Review of Current Challenges, Future Trends and Prospects for Emerging 5G-IoT Scenarios. *IEEE Access* 8 (2020), 23022–23040. https://doi.org/10.1109/ACCESS.2020.2970118
[41] Jiawei Shao and Jun Zhang. 2020. Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.
[42] Khe Chai Sim, Petr Zadrazil, and Fran?oise Beaufays. 2019. An Investigation Into On-device Personalization of End-to-end Automatic Speech Recognition Models. arXiv:eess.AS/1909.06678

[43] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[44] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A survey on deep transfer learning. In *International conference on artificial neural networks*. Springer, 270–279.
[45] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. 2021. To Bridge Neural Network Design and Real-World Performance:A Behaviour Study for Neural Networks. In *MLSys*.
[46] Surat Teerapittayanon, Bradley McDanel, and HT Kung. 2017. Distributed Deep Neural Networks over the Cloud, the Edge and End Devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 328–339.
[47] Charlott Vallon, Ziya Ercan, Ashwin Carvalho, and Francesco Borrelli. 2017. A machine learning approach for personalized autonomous lane change initiation and control. In *2017 IEEE Intelligent Vehicles Symposium (IV)*. 1590–1595. https://doi.org/10.1109/IVS.2017.7995936
[48] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. R. Alface, T. Bostoen, and F. De Turck. 2016. HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks. *IEEE Communications Letters* 20, 11 (2016), 2177–2180.
[49] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2018. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564* (2018).
[50] Praneeth Vepakomma, Tristan Swedish, Ramesh Raskar, Otkrist Gupta, and Abhimanyu Dubey. 2018. No Peek: A Survey of private distributed deep learning. arXiv:cs.LG/1812.03288
[51] Lingdong Wang, Liyao Xiang, Jiayu Xu, Jiaju Chen, Xing Zhao, Dixi Yao, Xinbing Wang, and Baochun Li. 2020. Context-Aware Deep Model Compression for Edge Cloud Computing. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 787–797.
[52] Chuhan Wu, Fangzhao Wu, Tao Di, Yongfeng Huang, and Xing Xie. 2020. FedCTR: Federated Native Ad CTR Prediction with Multi-Platform User Behavior Data. *arXiv preprint arXiv:2007.12135* (2020).
[53] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM Trans. Intell. Syst. Technol.* 10, 2, Article 12 (Jan. 2019), 19 pages. https://doi.org/10.1145/3298981
[54] Seunghyun Yoon, Hyeongu Yun, Yuna Kim, Gyu-tae Park, and Kyomin Jung. 2017. Efficient transfer learning schemes for personalized language modeling using recurrent neural network. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*.
[55] Pengpeng Zhao, Anjing Luo, Yanchi Liu, Fuzhen Zhuang, Jiajie Xu, Zhixu Li, Victor S Sheng, and Xiaofang Zhou. 2020. Where to go next: A spatio-temporal gated network for next poi recommendation. *IEEE Transactions on Knowledge and Data Engineering* (2020).
[56] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Asynchronous stochastic gradient descent with delay compensation. In *International Conference on Machine Learning*. PMLR, 4120–4129.
[57] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1059–1068.