Learning the Optimal Partition for Collaborative DNN Training with Privacy Requirements

Letian Zhang *Student Member, IEEE* Jie Xu *Senior Member, IEEE*

***Abstract*—With the growth of intelligent Internet-of-Things (IoT) applications and services, deep neural network (DNN) has become the core method to power and enable increased functionality in many smart IoT devices. However, DNN training is difficult to carry out on end devices because it requires a great deal of computational power. The conventional approach to DNN training is generally implemented on a powerful computation server; nevertheless, this approach violates privacy because it exposes the training data to curious service providers. In this paper, we consider a collaborative DNN training system between a resource-constrained end device and a powerful edge server, aiming at partitioning a DNN into a front-end part running on the end device and a back-end part running on the edge server to accelerate the training process while preserving the privacy of the training data. With the key challenge being how to locate the optimal partition point to minimize the end-to-end training delay, we propose an online learning module, called Learn-to-Split (L2S), to adaptively learn the optimal partition point on-the-fly. This approach is unlike existing efforts on DNN partitioning that relies heavily on a dedicated offline profiling stage. In particular, we design a new contextual bandit learning algorithm called LinUCB-E as the basis of L2S, which has provable theoretical learning performance and is ultra-lightweight for easy real-world implementation. We implement a prototype system consisting of an end device and an edge server, and experimental results demonstrate that L2S can significantly outperform state-of-the-art benchmarks in terms of reducing the end-to-end training delay and preserving privacy.**

***Index Terms*—Edge intelligence, deep learning, edge computing, online learning.**

I. Introduction

Deep neural networks (DNNs) have made tremendous progress towards the integration with a wide range of mobile and IoT applications such as face recognition, object detection and speech assistant. Nowadays, DNNs are increasingly being trained/updated using local and private data generated/collected by end devices (e.g., IoT devices and mobile devices, to suit the users' personal needs and quickly adapt to changing environment conditions. Although steps have already been taken recently to enable efficient DNN training on resource-constrained end devices [1], e.g., compressed/pruned models, lightweight deep learning frameworks such as TensorFlow Lite and PyTorch Mobile, and new-generation hardware, they face significant challenges to address the immediate deep learning needs of many existing end devices, which exhibit a substantial heterogeneity in terms of their computing capabilities. For instance, a Facebook study in 2019 showed that over 50% end devices were using processors before 2014, which limited what Facebook AI service could offer [2].

An alternative approach for DNN training for resource-constrained end devices is edge computing. Unlike centralized and datacenter-based cloud computing, edge computing allows the end device to use a nearby computing server located at the edge of the network, e.g., a cellular base station or a Wi-Fi hotspot, with ultra-low network latency. With edge computing, the end device can send its data to and train the DNN on an edge server, which is often much more computationally powerful than the end device itself. However, this approach also suffers an obvious drawback due to the full reliance on an external party to process private data – once an end device reveals its data to the edge server for DNN training, it becomes very hard to retain the full control of this data. While new cryptographic tools such as homomorphic computing [3] are being developed to enable computation on encrypted data, the complexity of DNN training hinders the immediate adoption of these methods to enable privacy-preserving DNN training in edge computing.

To utilize the computing power of the edge server while protecting user data privacy, collaborative DNN training between the end device and the edge server has attracted increasing attention recently [4]–[6]. The idea is to partition the DNN into a front-end part running on the end device and a back-end part running on the edge server. The end device feeds its raw data into the DNN, processes it through a certain layer, and then sends the intermediate data over the wireless network to the edge server to complete the feed-forward stage. Likewise, the back-propagation stage starts with the edge server updating parameters in the back-end part of the DNN; the intermediate gradients are then sent back to the end device to finish the whole training cycle. At the core of collaborative DNN training is a DNN partitioning problem, namely deciding at which partitioning point to split the DNN. Solving this problem needs to conquer several key challenges as follows.

Data Privacy Constraint. By sending the intermediate data instead of the raw data to the edge server, collaborative DNN training adds a layer of data privacy protection when offloading part of computation burden to the edge server. In fact, collaborative DNN training can be seen as a "soft" cryptographic method, which uses the front-end feed-forward DNN to "encode" the raw input data. However, such a "soft" cryptographic method is not perfect as existing works have shown that it is possible to reconstruct the raw input data using the intermediate data if the adversary is given the DNN structure, and the reconstruction quality improves if intermediate data earlier in the DNN pipeline is used [7].

L. Zhang and J. Xu are with the Department of Electrical and Computer Engineering, University of Miami. Email: {lxz437, jiexu}@miami.edu. This work is supported in part by NSF under grants 2006630, 2033681, 2029858 and 2044991.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

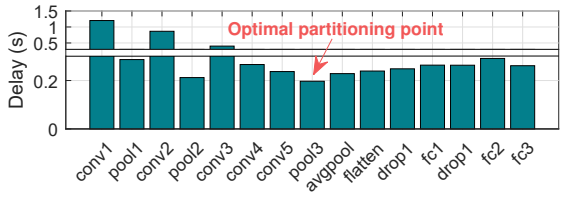


Fig. 1. End-to-End training delay at different partition points under 100 Mbps wireless network (AlexNet).

Therefore, the desired data privacy level limits the size of the back-end DNN and hence the amount of training workload that can be offloaded to the edge server.

Varying Network Environment. Different DNN partitioning points lead to different front-end workloads, intermediate data sizes, and back-end workloads. Assuming given system parameters such as the device processing speed, the wireless transmission rate and the edge processing speed, the optimal DNN partitioning point can be derived by solving a simple optimization problem to minimize the overall training latency. For example, Fig. 1 illustrates the overall training latency of different partitioning points of AlexNet for a given set of system parameters, where *pool3* is the optimal partitioning point in this case. However, the network environment in practice can often change, thereby rendering any static solution suboptimal. For example, the network transmission rate can change due to the dynamic spectrum management of the wireless carrier, the multi-user interference, and the mobility of end devices; the edge server processing capability may also change over time due to the edge server resource management to support multi-tenancy or even the change of edge servers due to location change.

Limited Feedback. To adapt to the changing network environment, existing works propose to utilize real-time system parameters as input. These system parameters, however, not only are ever-changing, but also can be very difficult, if not impossible, for an end device to obtain in practice. As is often the case, the end device can observe only the overall delay between feeding in the raw data and finishing one batch of training, but is unable to accurately decompose this delay into different components (e.g., transmission delay and training delay at the edge). This limited feedback challenge is similar to the congestion control problem in the classic Transport Control Protocol (TCP), where the end-user adjusts its congestion window based on only a binary congestion signal from the network as a summary of all network effects. With limited feedback, solutions that rely on explicit real-time system parameters become infeasible.

Model Synchronization. Splitting a DNN into a front-end part and a back-end part running on the end device and the edge server, respectively, has been mainly studied in the context of performing DNN inference tasks [8]–[11], including our prior work [12]. However, training DNN models based on the DNN partitioning idea needs to address new challenges as the DNN model is constantly being updated while the DNN model in inference tasks remains constant. Since the DNN partitioning point may change over time to adapt to new environments, extra data transmissions are needed to synchronize the DNN models between the end device and the edge server, thereby creating a new trade-off between the adaptation rate and the model synchronization cost.

In this paper, we design and build a collaborative DNN training system based on DNN partitioning, which aims to minimize the training delay under a data privacy constraint. A key component of our system is a novel online learning algorithm, called Learn-to-Split (L2S), that automatically learns the optimal DNN partitioning point based on the limited training delay feedback. Specifically, our main contributions are as follows:

(1) **Framework and algorithm**: We formulate the collaborative DNN training problem and propose the L2S algorithm to learn the optimal DNN partition. L2S addresses the aforementioned challenges by performing online learning using limited training delay feedback. It requires ultra-lightweight computation and minimal storage and hence, it is easy to deploy in practical systems. The core of L2S is a modified linear bandit learning algorithm, which addresses unique problems in collaborative training systems and has a provable performance guarantee.

(2) **Prototype and experiment**: We build a prototype collaborative DNN training system, where a Nvidia Jetson TX2 device, a fair representation of end devices, collaboratively trains a DNN with a GPU-powered edge server via a wireless link. We conduct extensive experiments to evaluate the performance of the proposed L2S algorithm. The results show that L2S is able to accurately learn the optimal partition point, and hence accelerates DNN training for various DNN model structures and under various wireless networks while ensuring the desired level of privacy protection.

II. Related work

A. Deep learning on end devices

Fully on-device learning. Deep learning for end devices has become a hot topic [13], covering hardware architecture, computing platforms, and algorithmic optimization. Many CPU/GPU vendors are developing new processors to support tablets and smartphones to run DL empowered applications, a notable example being Apple Bionic chips [14]. To support on-device neural network training, one approach is to quantize the weights and/or activations of a DNN model into lower-bit representation [15], [16]. Another common approach is to directly hand-craft more efficient mobile architecture [17]–[19]. However, these techniques are unlikely to address the immediate needs of all existing end devices, especially low-end and legacy devices that can not fully benefit from new computing architectures.

Fully offloading-based learning. Multi-access edge computing [20] enables cloud computing capabilities and an IT service environment at the edge of the cellular network. By offloading data, running applications and performing related processing tasks closer to the end-users, network congestion is reduced and applications perform better. Some efforts have been made to migrate the DNN training to edge servers [20]–[22]. However, this means that a large amount of raw data should be uploaded to the edge servers, not only causing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

TABLE I
COMPARISON OF DNN PARTITIONING METHODS

	Inference or Training	Edge Info	Privacy	Delay
Neurosurgeon [8] IONN [9] Edgent [10] DADS [11]	Inference	Offline	No	Yes
JointDNN [23]	Inference & Training	Offline	No	Yes
Hiertrain [24]	Training	Offline	No	Yes
Arden [4] DP-A [5] Siamese [6]	Training	Offline	Yes	No
ANS [12]	Inference	Online	No	Yes
L2S	Training	Online	Yes	Yes

TABLE II
KEY NOTATIONS

Notations	Definitions
p	Partition point
t	Training round
ψ_p^f	Intermediate forward result
ψ_p^b	Intermediate backward gradient
$d_p^{\mathtt{front,f}}$,	Front-end forward delay
$d_p^{\mathtt{front,b}}$	Front-end backward delay
$d_p^{\mathtt{back,f}}$,	Back-end forward delay
$d_p^{\mathtt{back,b}}$	Back-end backward delay
$d_p^{\mathtt{f,tx}}$	Forward transmission delay
$d_p^{\mathtt{b,tx}}$	Backward transmission delay
d_p^m	End device training delay
d_p^e	Edge training delay
P^c	Desired level of data privacy

prohibitive communication overhead, but also representing a huge privacy risk to owners of the end devices. By contrast, this paper does not simply migrate the DNN training service to the edge servers but more importantly investigates an online learning-based DNN partitioning method to accelerate DNN training performance as well as preserve the privacy of training data.

Partitioning-based collaborative learning. As summarized in Table I, several existing works have investigated the powerful delay improvements or privacy-preserving on collaborative deep learning intelligence [4]–[6], [8]–[12], [23], [24]. In [4]–[6], authors investigate the privacy-preserving of collaborative DNN training on mobile edge computing system. However, these works ignore the effect of DNN partition on training delay and some inappropriate partition points can greatly increase training delay. In [8]–[11], the authors study the collaborative DNN inference system via DNN partitioning. However, our work focuses on collaborative DNN training, which is very different from inference as backward propagation also needs to be executed in every training round. There are two prior works that study training delay improvements of collaborative DNN training on mobile edge systems [23], [24]. In [23], the authors assume that the whole backward propagation is executed on the edge server, and in each time slot the end device downloads the front partition part from the edge server based on the optimal partition point. In [24], a hybrid parallelism method is proposed to adaptively assign the DNN model layers across the three-level workers, where the backward propagation phase requires two weak workers to send their gradients to the main work for averaging the gradients. All these works require an offline profiling phase to measure the network condition, the processing ability of the end device, and the computing capacity of the edge server. Given this information, the optimal partition point is determined by solving an optimization problem. However, the knowledge acquired during offline profiling can be easily outdated considering the highly dynamic environment in MEC systems and frequently updating the knowledge will incur large overhead. By contrast, our method uses online learning to learn the optimal partition on-the-fly. Our prior work ANS [12] also aims to find optimal partition points by online learning. However, ANS is designed for DNN inference problem while the current paper studies a DNN training problem. Compared with DNN inference where the data flow is unidirectional from the end device to the edge server, DNN training consists of a forward propagation to compute the training loss as well as a backward propagation to update the model parameters. Therefore, the partitioning point models are different. More importantly, compared with DNN inference where the DNN model is given and fixed, in DNN training, the DNN model is constantly being updated. Because the DNN partitioning point may change over time to adapt to new environments, L2S has to specifically consider the effect of model synchronization and the trade-off between the adaptation speed and the synchronization cost. Moreover, the current paper takes data privacy into account when designing the partitioning strategy while ANS does not.

B. Contextual bandit learning

Multi-armed bandit (MAB) problem has been widely studied to address the critical tradeoff between exploration and exploitation in sequential decision making under uncertainty [25]. Contextual bandit learning extends the basic MAB by considering the context-dependent reward functions to estimate the relation of the observed context and the uncertain environment, where LinUCB is a classic algorithm [26]. However, the special on-device processing decision causes a difficult challenge for LinUCB, forcing them to stop learning the first time when on-device processing is selected. Our algorithm LinUCB-E innovatively incorporates a forced sampling technique to conquer this challenge, while still achieving provably asymptotically optimal performance.

III. SYSTEM ARCHITECTURE

A pictorial overview of our system architecture is given in Fig. 2. Next, we describe the architecture of the collaborative DNN training system. To make it clear, the key notations used in this paper are summarized in Table II.

A. Marking Partition Points

Let $\mathcal{P} = \{0, 1, 2, \ldots, P\}$ collect all potential DNN partition points. A partition point $p \in \mathcal{P}$ partitions a DNN into two parts: 1) the front-end part, $\mathtt{DNN}_p^{\mathtt{front}}$, contains layers from the input to the partition point $p \in \mathcal{P}$, and 2) the back-end

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

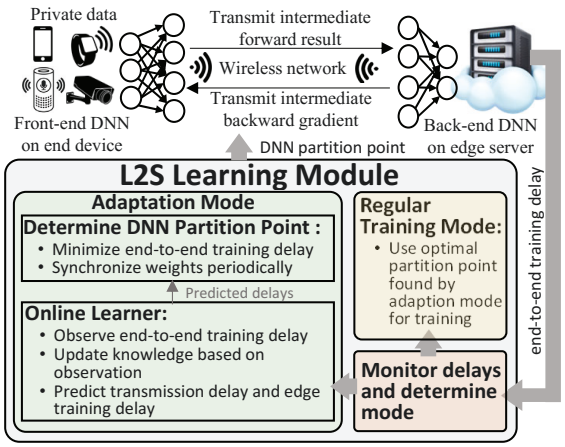


Fig. 2. An overview of the system architecture.

part, $\texttt{DNN}_p^{\texttt{back}}$, contains layers from the partition point p to the output layer. For example, if the partition point is placed at $p = 2$, then $\texttt{DNN}_p^{\texttt{front}}$ contains layers $\{1, 2\}$ and $\texttt{DNN}_p^{\texttt{back}}$ contains layers $\{3, 4, \ldots, P\}$. The partition points $p = 0$ and $p = P$ are the special cases: the partition $p = 0$ gives an empty $\texttt{DNN}_p^{\texttt{front}}$ which means the end device transmits raw input data to the edge server to train the entire DNN; the partition $p = P$ gives an empty $\texttt{DNN}_p^{\texttt{back}}$ indicating that all DNN layers are trained on the end device. Each DNN training round comprises of a forward propagation phase that stores all variables computed at each layer, and a backward propagation phase that calculates the gradient of weight parameters, which are used to update the DNN weights by stochastic gradient descent. The forward output of $\texttt{DNN}_p^{\texttt{front}}$ is called *the intermediate forward result* of partition p, denoted by ψ_p^f. Note that ψ_p^f contains the output of partition point p and ground truth labels, which will be sent to the edge server for the remaining forward processing. The backward gradient of $\texttt{DNN}_p^{\texttt{back}}$ is called *the intermediate backward gradient* of partition p, denoted by ψ_p^b. We assume that ψ_p^f and ψ_p^b include necessary overhead for data packet transmission (e.g., packet header) and follow-up DNN merging (e.g., information about the partition point).

B. Breakdown of DNN Training Delay

The end-to-end collaborative DNN training delay consists of six main parts: (1) **Front-end forward delay** $d_p^{\texttt{front,f}}$ of $\texttt{DNN}_p^{\texttt{front}}$ on the end device; (2) **Forward transmission delay** $d_p^{\texttt{f,tx}}$ for transmitting the intermediate forward result ψ_p^f from the end device to the edge server; (3) **Back-end forward delay** $d_p^{\texttt{back,f}}$ of $\texttt{DNN}_p^{\texttt{back}}$ on the edge server; (4) **Back-end backward delay** $d_p^{\texttt{back,b}}$ of $\texttt{DNN}_p^{\texttt{back}}$ on the edge server; (5) **Backward transmission delay** $d_p^{\texttt{b,tx}}$ for transmitting the intermediate backward gradient ψ_p^b from the edge server to the end device; (6) **Front-end backward delay** $d_p^{\texttt{front,b}}$ of $\texttt{DNN}_p^{\texttt{front}}$ on the end device. The end-to-end training delay with partition point p is therefore

$$d_p = d_p^{\texttt{front,f}} + d_p^{\texttt{f,tx}} + d_p^{\texttt{back,f}} + d_p^{\texttt{back,b}} + d_p^{\texttt{b,tx}} + d_p^{\texttt{front,b}} + \eta$$

where η is a Gaussian random variable to model the randomness in the training and transmission processes.

Fig. 3. Intermediate results after each layer (AlexNet).

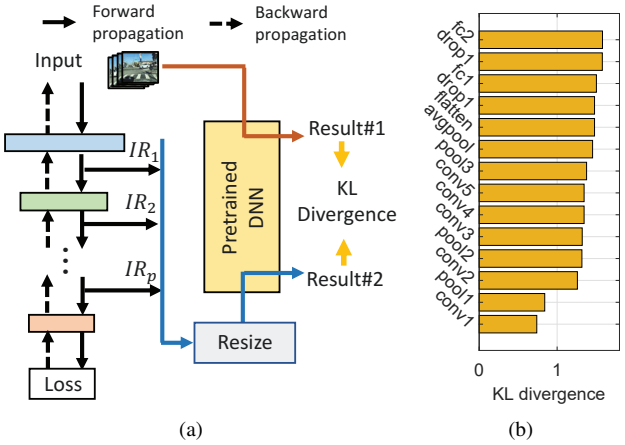


Fig. 4. KL divergence and example on AlexNet.

The forward and backward transmission delays $d_p^{\texttt{f,tx}}$, $d_p^{\texttt{b,tx}}$ are determined by the data size of the intermediate results ψ_p^f and ψ_p^b and the wireless uplink and downlink transmission rate, which vary depending on the network condition. The training delays $d_p^{\texttt{front,f}}$, $d_p^{\texttt{back,f}}$, $d_p^{\texttt{back,b}}$, $d_p^{\texttt{front,b}}$ of partitioned DNNs depend on many more factors: the number of DNN layers, the computational complexity of component layers, and also the processing speed of the end device/edge server. While some of them are fixed once the DNN structure is given (e.g. the number of layers and layer-wise computational complexity), others depend on the configuration of the computing platform which may also be time-varying (e.g. multi-user scheduling by the edge server).

We note that the configuration of the computing platform on the end device is relatively stable and fully revealed to the decision maker, i.e. the end device itself, and hence the front-end training delay $d_p^{\texttt{front,f}}$, $d_p^{\texttt{front,b}}$ of $\texttt{DNN}_p^{\texttt{front}}$ can be easily measured statistically for a given DNN using methods such as offline profiling. In the experiment, we use the application-specific profiling methods in [23] to obtain the expected training delay of $\texttt{DNN}_p^{\texttt{front}}$. Now, the key difficulty lies in learning $d_p^{\texttt{f,tx}} + d_p^{\texttt{back,f}} + d_p^{\texttt{back,b}} + d_p^{\texttt{b,tx}}$ for different partition points as a result of the unknown and time-varying edge computing capability and network condition. For ease of exposition, we define $d_p^m = d_p^{\texttt{front, f}} + d_p^{\texttt{front,b}}$ as the **end device training delay** and $d_p^e = d_p^{\texttt{f,tx}} + d_p^{\texttt{back,f}} + d_p^{\texttt{back,b}} + d_p^{\texttt{b,tx}}$ as the **edge training delay**.

C. Collaborative DNN Training

In collaborative DNN training, the end device owns the training data set (including samples and labels) while the edge server does not. Both the end device and the edge server have the same pre-selected DNN architecture but its weights must be obtained via training. We consider a batch training setting,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

where a batch of data samples are trained in each training round (which contains a forward propagation phase and a back propagation phase), and use $\{1, ..., T\}$ to index the training round. The end device has to pick a partition point p_t to perform collaborative DNN training in every round.

Privacy constraints: DNN layers act as filters to extract task-relevant information and abandon the rest [27]. As shown in Fig. 3, when we proceed through the deep network layers, the feature becomes more specific to the main task and irrelevant information (including sensitive information) will be gradually lost. Therefore, reconstructing the original image becomes much harder [28], which means that the end device's data privacy is better protected. Therefore, the end device's privacy requirements will limit the feasible set of DNN partitioning points. To enable a direct comparison with existing privacy-preserving schemes, we quantify the privacy loss using the Kullback-Leibler (KL) divergence [29]. Specifically, we extract intermediate results (IRs) at different partition points in the forward propagation and feed them into a pre-trained DNN to obtain the inference output. Then, we calculate the KL divergence between the inference outputs of original input and IRs. The rationale of this experiment is that, if the IR retains similar contents as the original input, then their corresponding inference outputs should have similar distributions, hence a smaller KL divergence. Thus, a larger KL divergence means a stronger privacy protection level. Fig. 4(a) illustrates the process of KL divergence calculation. Fig. 4(b) reports the computed KL divergence values on AlexNet and confirms our intuition that it becomes larger as the partitioning point is later in the DNN.

Objectives: The goal of collaborative DNN training is to minimize the training delay while satisfying the privacy constraint of the end device. Specifically, the end device aims to choose the optimal partition point in every collaborative DNN training round to minimize the end-to-end training delay under the data privacy constraint. Thus, the problem can be formulated as follows:

$$\begin{aligned} &\min_{p} \ \ d_p^{\mathrm{m}} + d_p^{\mathrm{e}} \\ &s.t. \ \ P^c \leq p \leq P \end{aligned} \tag{1}$$

where the constraint in (1) indicates that the partition point p should be no earlier than a given layer P^c for a desired level of data privacy protection.

Challenges: The above problem can be solved easily if the dependency of d_p^{m} and d_p^{e} on the partition point p is known. However, as aforementioned, d_p^{e} can easily change due to the time-varying edge computing capability and network condition. Therefore, the optimal partition point has to be learned on the fly.

Another practical consideration is that the weights of DNN models on the end device and the edge server need to be synchronized when the partition point changes. For example, as shown in Fig. 5, suppose at training round $t-1$, the partition point is $p = 2$ in a 5-layer DNN. This means that layers $\{1, 2\}$ are the front-end DNN on the end device while layers $\{3, 4, 5\}$ constitute the back-end DNN on the edge server. Suppose at training round t, the partition point changes to $p = 3$ to adapt to the changing environment, thereby moving layer 3 to the front-end. Because the model parameters are updated after every batch training, the current weights of layer 3 must also be sent to the end device.

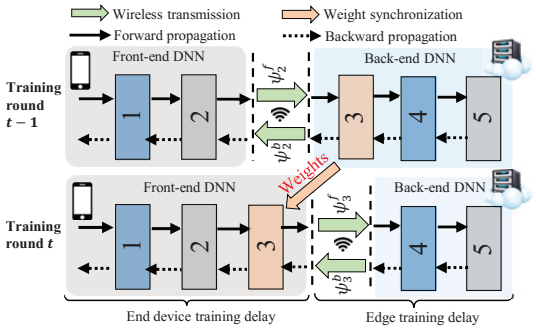


Fig. 5. An illustration of weight synchronization.

IV. Adaptive DNN Partitioning

In this section, we describe the design of the online learning module, called Learn-to-Split (L2S), in the collaborative DNN training system.

A. Edge Training Delay Prediction

To obtain the edge training delay d_p^e as a function of the partition point p, our idea is to learn a prediction model that maps contextual features of a partition point to the edge training delay. Since learning works online, this prediction model updates itself using the limited feedback information about the past observed d_p^e to closely follow the changes in the unknown system parameters. Using contextual features of the partition points has a clear advantage over learning the edge training delay of individual partition points separately, especially when possible partition points are many. This is because the underlying relationship between different partition points is captured by their contextual features, and hence, knowledge gained by choosing one particular partition point can be easily transferred to learning about the performance of all other unselected partition points.

*1) **Contextual Features of Partitions**:* We first construct contextual features associated with $\mathtt{DNN}_p^{\mathtt{back}}$ that may impact the back-end forward delay $d_p^{\mathtt{back,f}}$. Intuitively, the back-end forward delay is linearly dependent on the computation complexity of the back-end partition $\mathtt{DNN}_p^{\mathtt{back}}$, which usually is captured by the number of multiply-accumulate (MAC) units contained in $\mathtt{DNN}_p^{\mathtt{back}}$. It was shown in [30] that MAC units take up more than 90% of the execution time. Our experiment shows that the required computation time for one MAC unit is in fact different for different types of DNN layers. This is because different DNN layers allow different levels of parallel computation, especially when GPU is involved in the computation process. Since different partition points result in different combinations of layer types in $\mathtt{DNN}_p^{\mathtt{back}}$, simply using the total number of MAC units to predict $d_p^{\mathtt{back,f}}$ is problematic. To address this issue, instead of using a single scalar value for the total number of MAC units, we calculate the number of MAC units for each layer type, and use this vector for learning

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

the back-end forward delay. Specifically, we consider three main types of layers in DNN: *i*) convolutional layer, *ii*) fully-connected layer, and *iii*) activation layer, denoted by $m_p^{\mathtt{c}}$, $m_p^{\mathtt{f}}$ and $m_p^{\mathtt{a}}$, respectively.

For the transmission delay $d_p^{\mathtt{f},\mathtt{tx}}$ and $d_p^{\mathtt{b},\mathtt{tx}}$, although the wireless uplink and downlink rate may be unknown, it is still clear that $d_p^{\mathtt{f},\mathtt{tx}}$ and $d_p^{\mathtt{b},\mathtt{tx}}$ depend linearly on the data size of the intermediate results ψ_p^f and ψ_p^b, respectively.

The backward propagation involves the chain-rule gradient computation and the weight parameter updating. According to [31], the complexity of the gradient computation also depends on the MAC units of the DNN so we reuse $m_p^{\mathtt{c}}$, $m_p^{\mathtt{f}}$ and $m_p^{\mathtt{a}}$ as contextual features to predict the backward propagation delay. In addition, the weight parameter updating complexity is linear in the number of neurons in the different types of DNN layers. Therefore, we also use the number of weights of convolutional layers $n_p^{\mathtt{c}}$, fully-connected layers $n_p^{\mathtt{f}}$, and activation layers $n_p^{\mathtt{a}}$ in $\mathtt{DNN}_p^{\mathtt{back}}$ as additional contextual features.

To sum up, the contextual feature of a partition point p is collected in $\boldsymbol{x}_p = [m_p^{\mathtt{c}}, m_p^{\mathtt{f}}, m_p^{\mathtt{a}}, \psi_p^f, \psi_p^b, n_p^{\mathtt{c}}, n_p^{\mathtt{f}}, n_p^{\mathtt{a}}]^\top$. Here, we slightly abuse notation to use ψ_p^f and ψ_p^b to denote the data sizes of the IRs.

2) Linear Prediction Model: Although the best model for predicting the edge training delay is unclear due to the obscured DNN training process, we adopt a linear model due to the reasons mentioned above. In addition, compared to the other more complex and non-linear prediction models (such as a neural network), the linear model is much simpler and requires minimal resources on the end device. We show later in the experiments that this linear model is in fact validated to be a very good approximation.

Specifically, our prediction model has the form $d_p^e = \boldsymbol{\theta}^\top \boldsymbol{x}_p$, where $\boldsymbol{\theta}$ is the linear coefficients to be learned, which captures the unknown effects of the unknown system parameters (i.e., wireless conditions, computation capability of the edge server) on the edge training delay performance. In runtime, the coefficients will be updated online as new observations of d_p^e as a result of the partition decision p are obtained.

B. Online Learning Algorithm

Since we adopt a linear prediction model above, LinUCB [32], a classic online learning algorithm for linear models that gracefully handles the exploitation v.s. exploration trade-off, seems a good candidate for solving our problem. However, LinUCB has a significant limitation for it to work effectively in our system. In what follows, we first explain how LinUCB works and its limitation in L2S. Next, we propose a new learning algorithm that overcomes this limitation.

1) LinUCB and its Limitation: The basic idea of LinUCB is an online linear regression algorithm, which incrementally updates the linear coefficients using newly acquired feedback. However, when making decisions, LinUCB takes into account the confidence of the prediction for different actions' expected payoff (i.e., the delay of different partition points in our case). Put in the context of DNN partition, LinUCB maintains two auxiliary variables $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ and $\boldsymbol{b} \in \mathbb{R}^{d\times 1}$ for estimating the coefficients $\boldsymbol{\theta}$. For each learning round t (where a learning round corresponds to one batch training), $\boldsymbol{\theta}$ is estimated by $\hat{\boldsymbol{\theta}}_t = \boldsymbol{A}_{t-1}^{-1}\boldsymbol{b}_{t-1}$, and the partition point for batch t is selected to be

$$p_t = \underset{\boldsymbol{p}\in\mathcal{P}}{\arg\min}\ d_p^{\mathtt{m}} + \hat{\boldsymbol{\theta}}^\top \boldsymbol{x}_p - \alpha\sqrt{\boldsymbol{x}_p^\top \boldsymbol{A}_{t-1}^{-1}\boldsymbol{x}_p} \tag{2}$$

In the function to be minimized, the first term $d_p^{\mathtt{m}}$ is the end device training delay of partition point p, which is assumed to be known; the second term $\hat{\boldsymbol{\theta}}^\top \boldsymbol{x}_p$ is the predicted edge training delay of partition point p using the current estimate $\hat{\boldsymbol{\theta}}$; the last term $\alpha\sqrt{\boldsymbol{x}_p^\top \boldsymbol{A}_{t-1}^{-1}\boldsymbol{x}_p}$ represents the confidence interval of the edge offloading training delay prediction. A larger confidence interval indicates that the prediction is less accurate and hence, even if the predicted delay of a partition point p is low, the chance to select this partition point should be lowered. After the training round is completed and the realized edge training delay $d_{p_t}^{\mathtt{e}}$ is observed, the auxiliary variables are updated as $\boldsymbol{A}_t \leftarrow \boldsymbol{A}_{t-1} + \boldsymbol{x}_{p_t}\boldsymbol{x}_{p_t}^\top$ and $\boldsymbol{b}_t \leftarrow \boldsymbol{b}_{t-1} + \boldsymbol{x}_{p_t} d_{p_t}^{\mathtt{e}}$.

However, LinUCB has a major limitation for it to work effectively in our online learning module. Among all possible partition points, the partition point $p = P$, namely pure on-device processing, is actually a very special partition point that does *not* follow the linear prediction model. This is because the edge training delay is always 0 once $p = P$ is selected and *any* linear coefficient is a "correct" coefficient since the contextual feature associated with $p = P$ is a zero vector. If, for some training rounds, $p = P$ is selected by LinUCB for training, then the auxiliary variables A_t and b_t do not get updated since there is no feedback/new information about the edge offloading delay. As a result, LinUCB will select $p = P$ according to the selection rule (2) for the next training round and thereafter, essentially being forced to stop learning and trapped in pure on-device training for all future training rounds. Therefore, LinUCB fails to work in our online learning module.

2) LinUCB with Escaping: In light of the limitation of LinUCB, we propose a modified online learning algorithm, called LinUCB-E, to support L2S. To escape from being trapped in pure on-device processing, a natural idea is to add randomness in the partition point selection. Because partition points other than the pure on-device processing have a chance to be selected, new knowledge about the edge offloading delay and hence $\boldsymbol{\theta}$ can be acquired. Our implementation of this randomness idea is through a forced sampling technique. Specifically, for a total number of T training rounds, we define a forced sampling sequence $\mathcal{F} = \{t | t = nT^\mu, t \le T, n = 1, 2, \dots\}$, where μ is a design parameter. If the training round index t belongs to $\mathcal{F}$, then LinUCB-E forces L2S to sample a partition point other than $p = P$. In other words, $p = P$ is not an option for these training rounds. According to the design of the sequence, forced sampling occurs every T^μ training rounds. Note that, forced sampling has no effect on training rounds when $p = P$ is not the selected partition point if the classic LinUCB were applied. Fig. 6 illustrates the idea of forced sampling. The pseudocode of LinUCB-E is given in Algorithm 1.

The parameter μ is a critical parameter of LinUCB-E, which controls the frequency of forced sampling. In the theorem

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

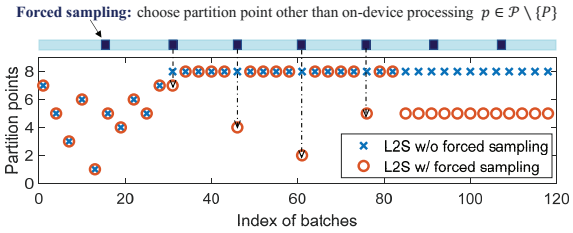


Fig. 6. Forced sampling: forced sampling is activated only when the partition decision are to be on-device processing.

Algorithm 1 LinUCB-E algorithm

Set possible partition domain $\mathcal{P}' = \{P^c, \ldots, P\}$
Construct x_p for candidate partition points $\forall p \in \mathcal{P}'$
Obtain front-end training delay estimate $d_p^{\texttt{front}}, \forall p \in \mathcal{P}'$
Determine forced sampling sequence $\mathcal{F}$.
Initialize $\boldsymbol{A}_0 = \beta I_d$, $b_0 = \mathbf{0}$.
for each batch $t = 1, 2, \cdots, T$ **do**
  Compute current estimate $\hat{\boldsymbol{\theta}}_t = \boldsymbol{A}_{t-1}^{-1} b_{t-1}$.
  for each candidate partition point $p \in \mathcal{P}'$ **do**
    Compute $\hat{d}_p^{\texttt{e}} = \hat{\boldsymbol{\theta}}_t^\top \boldsymbol{x}_p - \alpha\sqrt{\boldsymbol{x}_p^\top \boldsymbol{A}_{t-1}^{-1} \boldsymbol{x}_p}$.
  end for
  if $t \in \mathcal{F}$ **then**
    Choose $p_t = \arg\min_{p \in \mathcal{P}'_{\{\neq P\}}} d_p^{\texttt{front}} + \hat{d}_p^{\texttt{e}}$.
  else
    Choose $p_t = \arg\min_{p \in \mathcal{P}'} d_p^{\texttt{front}} + \hat{d}_p^{\texttt{e}}$.
  end if
  if $p_t \neq P$ **then**
    Observe $d_{p_t}^{\texttt{e}}$ once one batch training is done.
    $\boldsymbol{A}_t \leftarrow \boldsymbol{A}_{t-1} + \boldsymbol{x}_{p_t} \boldsymbol{x}_{p_t}^\top$, $\boldsymbol{b}_t \leftarrow \boldsymbol{b}_{t-1} + \boldsymbol{x}_{p_t} d_{p_t}^{\texttt{e}}$.
  else
    $\boldsymbol{A}_t = \boldsymbol{A}_{t-1}$, $\boldsymbol{b}_t = \boldsymbol{b}_{t-1}$.
  end if
end for

below, we characterize what is a good choice of μ and the resulting performance of LinUCB-E.

Theorem 1. Under mild technical assumptions, the regret (i.e., the training delay difference compared to an orcale algorithm that selects the optimal partition point for all T training rounds) of LinUCB-E, denoted by $R(T)$, satisfies: $\forall \delta \in (0, 1)$, with probability at least $1 - \delta$, $R(t)$ can be upper bounded by

$$\max\{O(T^{0.5+\mu} \log(T/\delta)), O(T^{1-\mu})\}$$

Proof. The proof is given in Appendix **??**. □

According to Theorem 1, by choosing $\mu \in (0, 0.5)$, the regret bound is sublinear in T, implying that the average end-to-end training delay asymptotically achieves the best possible end-to-end training delay when $T \to \infty$. For a finite T, this bound also gives a characterization of the convergence speed of LinUCB-E. In addition, by choosing $\mu = 0.25$, the order of the regret bound is minimized at $O(T^{0.75} \log(T))$.

C. DNN Model Synchronization

Since LinUCB-E may choose different partition points across training rounds to learn the parameter $\boldsymbol{\theta}$, it may cause weight inconsistency of the DNN layers that change processing locations (i.e., from edge to device, or from device to edge). A straightforward solution is to synchronize the weights of these layers between the end device and the edge server whenever the partition point changes. However, frequent weight synchronization will cause a large wireless transmission overhead, which increases the overall training delay. Note that model synchronization is not needed for partition-based DNN *inference* tasks because the adopted DNN model was pre-trained and does *not* change during inference.

To reduce the overhead due to model synchronization, we perform model synchronization only periodically. For the layers that change location due to adaptive partitioning, our system simply uses the old model weights for training until the next model synchronization. Apparently, the synchronization frequency has a trade-off between the synchronization overhead and the training performance: a higher frequency causes a larger overhead but less disturbance to the regular training process. Since theoretical results on the optimal synchronization frequency seem extremely hard, we design an experiment to derive an empirical understanding of what a good model synchronization frequency should be. Specifically, we consider the average change in the model weights in different layers after different numbers of training rounds compared to the initial model. As shown in Fig. 7, the weight change after 100 training rounds only accounts for a very small fraction ($\sim$0.05%) of the weight change after the accuracy reaches 97%. Thus, the model weight change for a small number of training rounds is almost negligible. In Fig. 8, we further show that the accuracy and the delay of training ResNet34 with different synchronization rates after 50000 total training rounds. We observe that a large synchronization period can significantly reduce the total training time but causes a lower training accuracy. Based on these experimental results, we decide to use 100 rounds as the synchronization period because it achieves a similar training accuracy as that when weights are synchronized in every training round, while reducing over 70% weight synchronization overhead.

D. L2S

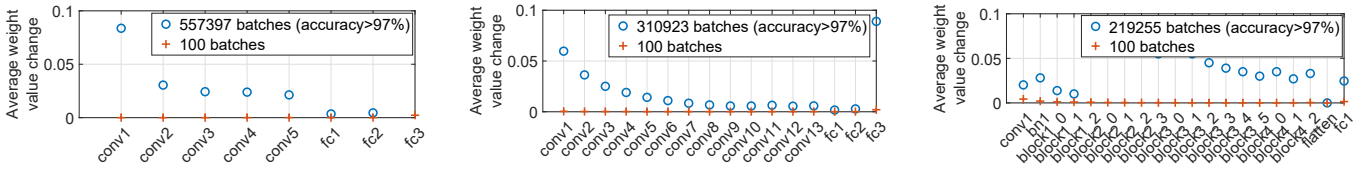
Now, we combine the components described so far to present the overall architecture of L2S.

In the runtime, L2S operates in either one of the following two modes: 1) Adaptation mode; 2) Regular training mode. A delay monitor is also deployed to trigger the switch between these two modes. We describe their functions as follows.

*1) **Adaptation mode**:* In the adaptation mode, L2S actively learns the new optimal partition point in a new network environment by running LinUCB-E. When entering this mode, L2S first synchronizes weights between the front-end part and the back-end part and then synchronizes weights only periodically. After a fixed number of T training rounds, L2S will check the history of partition decisions. If the partition decisions from $T - n_{th}$ to T are the same, L2S exits the adaptation mode.

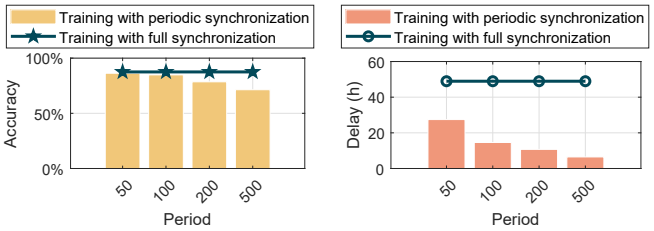
*2) **Regular training mode**:* For most of the time, L2S runs in the regular training model where the optimal partition point

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal



(a) AlexNet (b) Vgg16 (c) ResNet34

Fig. 7. Per partition part average weight value change caused by training.



(a) (b)

Fig. 8. The training accuracy and delay comparison of the periodic weight synchronization and full weight synchronization. We run training with different weight synchronization periods on ResNet34.

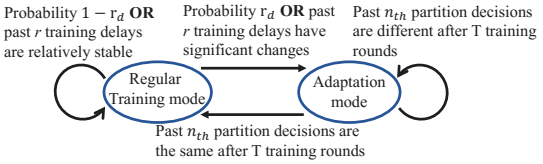


Fig. 9. Mode switching policy.

found by LinUCB-E in the previous adaptation mode is used to perform collaborative DNN training. When to enter a new adaptation mode is triggered by either one of the following events.

- Concept drift: a delay monitor constantly keeps track of the end-to-end training delay. If the end-to-end training delay in the past r training rounds experienced a significant change (determined by a user-defined threshold r_{th}), then L2S enters the adaptation mode to adapt to the new environment.
- Random: even without a concept drift, L2S enters the adaptation model with a probability r_d.

The mode switching of L2S is illustrated in Fig. 9.

V. Evaluation

In this section, we conduct experiments to show L2S's behavior and effectiveness.

A. Experiment Setup

Testbed setup. We use Nvidia Jetson TX2 as the end device, which contains a mobile SoC (Tegra TX2) with a shared 8 GB 128 bit LPDDR4 memory between GPU and CPU. The edge server is a PC equipped with an Intel Core i7-8700K CPU and a Nvidia GeForce GTX 1080 Ti GPU. The end device and edge server are connected via a point-to-point Wi-Fi, and we use NetLimiter4 [33] to set the wireless transmission speed to emulate different network conditions.

DNN models and dataset. Three state-of-the-art DNNs, namely AlexNet [34], Vgg16 [35] and ResNet34 [36] are considered in the experiment. We implement L2S using PyTorch, a popular machine learning platform, and run deep learning on these models and perform DNN partitioning. In order to get the intermediate gradient for backward propagation, we add hooks (via `register_hook` function provided by PyTorch) to backward propagation so that the intermediate gradient will be called after each gradient tensor is ready; we also set the parameter `requires_grad` in PyTorch DNN model to lock and unlock the weight updating of each layer. We use Netscope Analyzer [37], a web-based tool, for visualizing and analyzing DNN network architectures. For chain topology DNNs, we mark a partition point after each layer. However, it should be noted that some DNN models are not chain topology, in which case the residual block method [38] can be used to determine the partition points (e.g., ResNet34 has 16 concatenated residual blocks). We sample 10 classes' images from ImageNet [39] and use them as the training data. We set the training batch size to be 16 and the privacy requirement P^c to be 4.

Benchmarks. For the purpose of performance comparison, the following benchmarks are used in the evaluation of L2S.

- Oracle: Oracle selects the optimal partition point in each training round. We obtain the Oracle decision by measuring the performance of all possible partition points for 100 times and then picking the partition point with the minimum average delay.
- Fixed Partition Training (FPT): The partition point is fixed at layer P^c. The end device and the edge server execute the collaborative training at this fixed partition point.
- Pure On-Device Training (PDT): The end device trains the DNN model by itself without offloading.
- L2S with full weight synchronization (L2S-FWS): The weights are synchronized between the end device and the edge server whenever the partition point changes.

B. Experimental Results

1) Delay Prediction Error and Learning Convergence: Fig. 10 shows how the delay prediction error evolves as the number of training rounds increases. In this set of experiments, three DNN models are trained in different network conditions (high speed network: 100 Mbps, low speed network: 60 Mbps). As can be seen, L2S achieves an excellent prediction performance and can accurately predict edge training delay in about 40 training rounds. Fig. 11 shows the average end-to-end training delay of three DNN models achieved by Oracle and L2S

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal

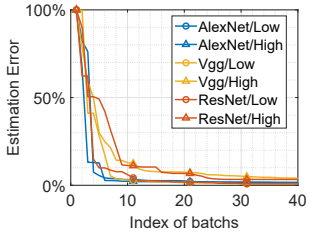


Fig. 10. Online prediction error reduces as more batches are analyzed.

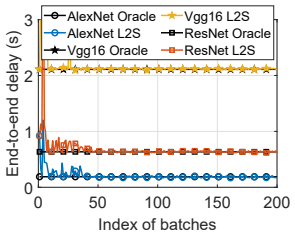


Fig. 11. End-to-end training delay reduces as more batches are analyzed.

at the low-speed network. The average end-to-end training delay of L2S quickly converges to that of Oracle in about 50 training rounds, starting from zero knowledge about the network condition and edge server capability.

2) End-to-end Training Delay Improvement: Fig. 12 shows the average per-round end-to-end training delay achieved by PDT, FPT and L2S under different network transmission speeds when the edge server uses GPU. When the transmission speed is low, the end-to-end training delay of L2S is close to PDT. This is because L2S tends to run the entire DNN on the end device to avoid large transmission delay for sending data to the edge server. When the transmission speed is moderate, L2S is able to make an effective trade-off between on-device processing and edge offloading. Although the end-to-end training delay of FPT decreases when the transmission speed is high, the end-to-end training delay of L2S still outperforms FPT. This is because that L2S can learn the network condition and find an optimal partition point to minimize the training delay. Fig. 12(d) summarizes the end-to-end training delay improvement in the best cases when the edge server uses CPU or GPU for all three DNNs. As it suggests, collaborative DNN training using L2S achieves a larger improvement when the edge server is more powerful.

3) Accuracy Performance: To illustrate the accuracy performance of L2S, we compare the training accuracy of L2S with L2S-FWS and PDT in Fig. 13(a) after 50000 training rounds. We also run L2S, L2S-FWS under two different network conditions (low speed network: 60 Mbps and high speed network: 100 Mbps), and report the total training time in Fig. 13(b). We use PDT as the baseline, and normalize the total training time of L2S, L2S-FWS with this baseline. As can be seen, L2S can reduce the total training time by more than 1/3, while the training accuracy only decreases about 2.14% on average.

4) Privacy Constraints: In this experiment, we change the privacy constraint P_c adopted by the users, and investigate its impact on the training performance achieved by L2S. As shown in Fig. 14, when the privacy requirement is higher (i.e. P_c is larger), the end-to-end training delay improvement achieved by L2S becomes smaller. This is because the feasible partition points become fewer, limiting what can be achieved by L2S.

Furthermore, we compare L2S with privacy-preserving training (PPT) in [5], which is proposed for DNN face recognition based on the Vgg16 model, to further demonstrate the effectiveness of L2S. In this case, we use the same method to calculate the KL divergence by using PPT (i.e., feed the IR produced by PPT into the pre-trained DNN). We set P^c, namely the privacy constraint in L2S, to a value so that its corresponding KL divergence is no less than that of PPT. In this way, we ensure that the privacy performance of L2S is at least as good as PPT. For a fair comparison, we run L2S and PPT on Vgg16 model with the same training data and compare the total training time and the KL divergence when the accuracy is higher than 97%. As can be seen in Fig. 15, L2S achieves a larger KL divergence than PPT by setting P^c to a desired level of data privacy protection, meanwhile reduces the total training time by more than 94%. This is because PPT takes the first convolutional layer of vgg16 as the partition point, whose intermediate data size is much larger than the input and IRs of the other layers. Therefore, when implemented in a wireless network with a poor wireless link, PPT incurs a significant training delay due to the slow data transmission. On the other hand, L2S chooses a proper partitioning point that usually has a smaller intermediate data size than the input. Therefore, L2S can significantly reduce the training time compared with PPT.

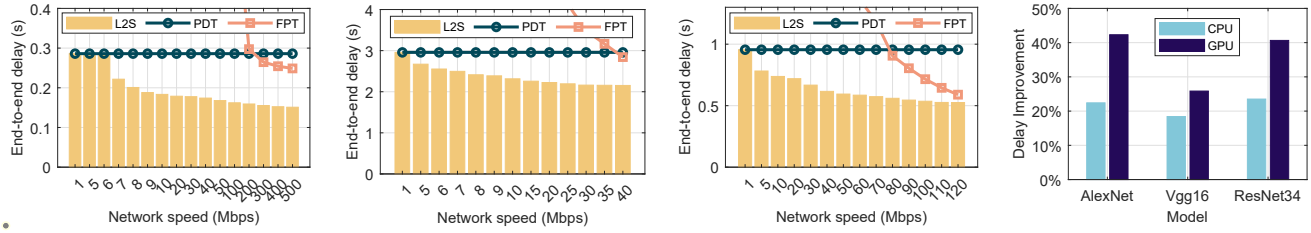
5) Forced Sampling: In this experiment, we investigate the impact of the forced sampling on L2S. We design two experiments as follows: One uses the LinUCB-E to find the optimal partition point (with forced sampling) and the other uses the original LinUCB to obtain the optimal partition point (without forced sampling). We test these two cases for 50 times and observe the average total training time on three DNN models, which is shown in Fig. 16. We can see that the average total training time of LinUCB (without forced sampling) is higher than that of LinUCB-E. This is because LinUCB sometimes will get trapped in the pure on-device processing and LinUCB-E can avoid this problem.

6) Dynamic Adaptation: Fig. 17 shows how L2S can track the change of the environment and adapt its partition point when the network condition changes on ResNet34. We plot L2S's per-round end-to-end training delay as the blue line and the network speed as the red line. We change the network speed from high to low at the training round 2000 and reset the network speed to high at training round 4000. By comparing the delay in the past 10 training rounds, the delay monitor determines an environment change and thus triggers L2S into the adaptation mode. Note also that the two adaptation modes indicated in the red boxes are due to the random mechanism in mode switching. Regardless of which mode L2S is in, the overall training accuracy keeps increasing during the entire training process.

VI. Conclusion

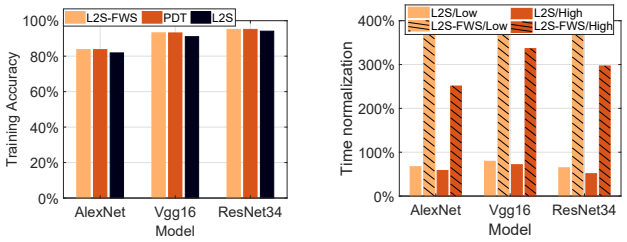
In this paper, we design and prototype a collaborative DNN training system on edge computing platforms, considering privacy requirements of the training data. We propose L2S, an integral component of the system that online learns the optimal DNN partition points, using only limited delay feedback without a dedicated offline profiling/training phase. L2S incorporates a simple yet effective forced sampling mechanism to ensure continued learning. To adapt to the dynamic environment, we introduce a delay monitor to decide the proper L2S

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3127715, IEEE Internet of Things Journal



(a) AlexNet (b) Vgg16 (c) ResNet (d) Best Reduction

Fig. 12. End-to-end training delay achieved by PDT, FPT and L2S, and delay reduction of L2S.



(a) (b)

Fig. 13. (a) Average accuracy of L2S, L2S-FWS and PDT with 50000 training rounds. (b) Total training time of L2S and L2S-FWS under different network conditions (PDT is the baseline).

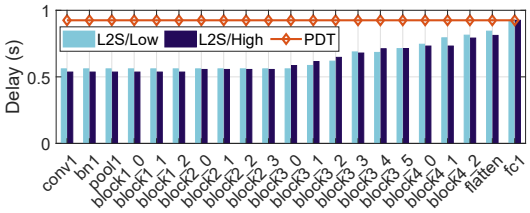


Fig. 14. Impact of privacy setting on the end-to-end training delay achieved by L2S (ResNet).

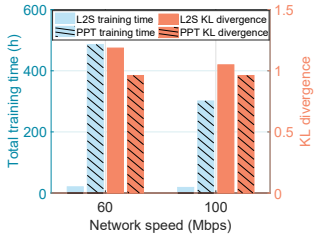


Fig. 15. Comparison of L2S and PPT.

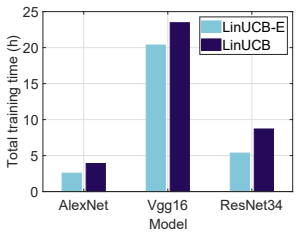


Fig. 16. Total training time of L2S with LinUCB-E and LinUCB.

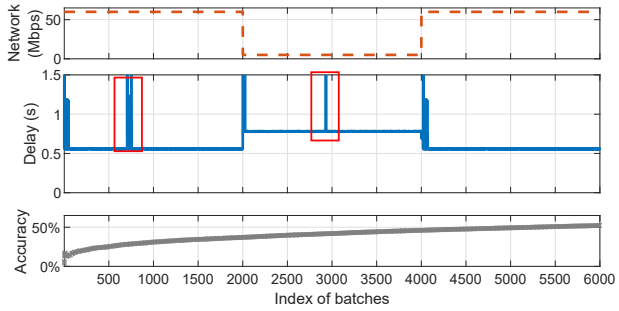


Fig. 17. L2S's reaction upon environment change.

mode. Our experiments show that L2S can significantly reduce the end-to-end training delay compared to pure on-device processing while keeping a comparable training accuracy. Together with existing efforts on accelerating deep learning on resource-constrained end devices, L2S will play an essential role in pushing the frontier of deep learning-empowered IoT intelligence. Currently, L2S studies the DNN partition between a single end device and an edge server on the training delay without specifically addressing the energy consumption. In our future work, we plan to extend the framework to the general setting where multiple end devices and multiple edge servers can be involved in training a DNN model, and consider the impact of energy consumption on the collaborative DNN training system.

REFERENCES

[1] K. Ota, M. S. Dao, V. Mezaris, and F. G. D. Natale, "Deep learning for mobile multimedia: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 13, no. 3s, pp. 1–22, 2017.

[2] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[3] F. Zhao, C. Li, and C. F. Liu, "A cloud computing security solution based on fully homomorphic encryption," in *16th international conference on advanced communication technology*. IEEE, 2014, pp. 485–488.

[4] J. Wang, J. Zhang, W. Bao, X. Zhu, B. Cao, and P. S. Yu, "Not just privacy: Improving performance of private deep learning in mobile cloud," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2407–2416.

[5] Y. Mao, S. Yi, Q. Li, J. Feng, F. Xu, and S. Zhong, "Learning from differentially private neural activations with edge computing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 90–102.

[6] S. A. Osia, A. S. Shamsabadi, S. Sajadmanesh, A. Taheri, K. Katevas, H. R. Rabiee, N. D. Lane, and H. Haddadi, "A hybrid deep learning architecture for privacy-preserving mobile analytics," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4505–4518, 2020.

[7] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1492–1499.

[8] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[9] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 401–411.

[10] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2019.

[11] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1423–1431.

[12] L. Zhang, L. Chen, and J. Xu, "Autodidactic neurosurgeon: Collaborative deep inference for mobile edge intelligence via online learning," in *Proceedings of the Web Conference 2021*, 2021, pp. 3111–3123.

[13] Y. Deng, "Deep learning on mobile devices: a review," in *Mobile Multimedia/Image Processing, Security, and Applications 2019*, vol. 10993. International Society for Optics and Photonics, 2019, p. 109930A.

[14] D. Sima. (2018) Apple's mobile processors.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[16] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[17] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[18] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[19] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[20] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[21] T. L. Duc, R. G. Leiva, P. Casari, and P.-O. Östberg, "Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–39, 2019.

[22] S. Shahzadi, M. Iqbal, T. Dagiuklas, and Z. U. Qayyum, "Multi-access edge computing: open issues, challenges and future perspectives," *Journal of Cloud Computing*, vol. 6, no. 1, p. 30, 2017.

[23] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, 2019.

[24] D. Liu, X. Chen, Z. Zhou, and Q. Ling, "Hiertrain: Fast hierarchical edge ai learning with hybrid parallelism in mobile-edge-cloud computing," *IEEE Open Journal of the Communications Society*, 2020.

[25] T. L. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[26] J. Langford and T. Zhang, "The epoch-greedy algorithm for contextual multi-armed bandits," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*. Citeseer, 2007, pp. 817–824.

[27] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[28] A. Dosovitskiy and T. Brox, "Inverting visual representations with convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4829–4837.

[29] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy, "Securing input data of deep learning inference systems via partitioned enclave execution," *arXiv preprint arXiv:1807.00969*, 2018.

[30] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices," in *Proceedings of the 25th ACM international conference on Multimedia*, 2017, pp. 1663–1671.

[31] E. Mizutani and S. E. Dreyfus, "On complexity analysis of supervised mlp-learning for algorithmic comparisons," in *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, vol. 1. IEEE, 2001, pp. 347–352.

[32] W. Chu, L. Li, L. Reyzin, and R. Schapire, "Contextual bandits with linear payoff functions," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 208–214.

[33] Netlimiter4. [Online]. Available: https://www.netlimiter.com/products/nl4.

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[37] Netscope analyzer. [Online]. Available: https://dgschwend.github.io/netscope/quickstart.htm.

[38] A. E. Eshratifar, A. Esmaili, and M. Pedram, "Bottlenet: A deep learning architecture for intelligent mobile cloud computing services," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2019, pp. 1–6.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

Letian Zhang received the B.S. degree in electrical engineering from Shanghai Normal University, Shanghai, China, in 2012 and the M.S. degree in electrical engineering from Shanghai University, Shanghai, China, in 2015. From 2015 to 2018, he worked in ZTE company as a Software Engineer. He is currently pursuing the Ph.D. degree with the College of Engineering, University of Miami. His primary research interests include mobile/IoT system design, edge intelligence, and network security.

Jie Xu (Senior Member, IEEE) received the B.S. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2008 and 2010, respectively, and the Ph.D. degree in electrical engineering from UCLA in 2015. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Miami. His research interests include mobile edge computing/intelligence, machine learning for networks, and network security. He received the NSF CAREER Award in 2021.