FedAdapt: Adaptive Offloading for IoT Devices in Federated Learning

Di Wu, Rehmat Ullah, Paul Harvey, Peter Kilpatrick, Ivor Spence, and Blesson Varghese

***Abstract*—Applying federated learning (FL) on Internet of Things (IoT) devices is necessitated by the large volumes of data they produce and growing concerns of data privacy. However, there are three challenges that need to be addressed to make FL efficient: 1) execution on devices with limited computational capabilities; 2) accounting for stragglers due to computational heterogeneity of devices; and 3) adaptation to the changing network bandwidths. This article presents FedAdapt, an adaptive offloading FL framework to mitigate the aforementioned challenges. FedAdapt accelerates local training in computationally constrained devices by leveraging layer offloading of deep neural networks (DNNs) to servers. Furthermore, FedAdapt adopts reinforcement learning (RL)-based optimization and clustering to adaptively identify which layers of the DNN should be offloaded for each individual device on to a server to tackle the challenges of computational heterogeneity and changing network bandwidth. The experimental studies are carried out on a lab-based testbed and it is demonstrated that by offloading a DNN from the device to the server FedAdapt reduces the training time of a typical IoT device by over half compared to classic FL. The training time of extreme stragglers and the overall training time can be reduced by up to 57%. Furthermore, with changing network bandwidth, FedAdapt is demonstrated to reduce the training time by up to 40% when compared to classic FL, without sacrificing accuracy.**

***Index Terms*—Edge computing, federated learning (FL), Internet of Things (IoT), reinforcement learning (RL).**

I. Introduction

INTERNET of Things (IoT) devices generate large volumes of data from user devices that are often deemed sensitive. For example, wearable devices, such as the Google Glass or Apple watch, gather sensitive data by recording the daily activities of users [1]. This data can be analyzed using machine learning (ML) techniques for delivering personalized services [2], [3]. Privacy preserving ML techniques are required to ensure that sensitive data can be analyzed in a safe manner.

Manuscript received 11 December 2021; revised 21 February 2022; accepted 13 May 2022. Date of publication 19 May 2022; date of current version 24 October 2022. This work was supported by Rakuten Mobile, Japan. The work of Blesson Varghese was supported by the Royal Society Short Industry Fellowship. *(Corresponding author: Di Wu.)*

Di Wu, Rehmat Ullah, and Blesson Varghese are with the School of Computer Science, University of St Andrews, St Andrews KY16 9AJ, U.K. (e-mail: dw217@st-andrews.ac.uk).

Paul Harvey is with the Autonomous Networking Research and Innovation Department, Rakuten Mobile, Tokyo 158-0094, Japan.

Peter Kilpatrick and Ivor Spence are with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT7 1NN, U.K.

Digital Object Identifier 10.1109/JIOT.2022.3176469

Federated learning (FL) is a privacy-preserving ML technique that has recently gained popularity [4]–[6]. Using this technique, an ML model, for example, a deep neural network (DNN), is executed on several IoT devices. The model on each device is trained without sending raw data (that may be sensitive) from the device to a server. Instead, the server receives intermediate models generated by the devices that are aggregated on the server to create a global model. Thus, an ML model can be trained by not exposing sensitive data from a device to an external server.

In the classic FL architecture, the computationally intensive workload (training of the DNN) is executed on the device. A server located at the edge of the network or on the cloud only aggregates the weights sent from the devices, which is relatively less computationally intensive. Aggregation is required for updating a global model on the server that is then sent to the devices to continue training. The devices train independently and may be connected to the server through various network configurations.

Classic FL is however limited in the following three ways.

1) *Impractical Training Times on Computationally Constrained IoT Devices:* The computationally intensive workload of training used in FL is required to be executed on devices that are relatively resource constrained when compared to large servers or clusters that may have specialized processors for training ML models. Thus, the time taken to train ML models on devices can be large making FL impractical for real-world scenarios [7]–[9]. For example, a lightweight convolutional neural network (CNN), MobileNetV1 [10], required over 8 h on Raspberry Pi3 single board computers to complete one round of training in FL [9]. Therefore, there is an immediate need for techniques that accelerate training in FL on IoT devices.
2) *Stragglers Arising From Computational Heterogeneity of IoT Devices Can Slow Down Other Devices During Training:* IoT devices connected to a server for FL may have varying computational capabilities or heterogeneous architectures. Devices that require a longer time for training, referred to as stragglers in this article, will slow down all devices in a synchronous FL system [11]–[13]. This is because the aggregating server will need to wait until all devices have completed training [14]. Asynchronous FL systems have been developed to mitigate the straggler problem. However, they affect the accuracy of the models since all devices may not contribute equally to training [15], [16]. Therefore,

approaches that reduce the impact of stragglers are required.

3) *Varying Operational Conditions Can Increase Training Time:* Operational conditions, such as the network bandwidth between a device and the server, can vary during the course of training. This can impact the training time [11], [17]. These need to be considered for efficient FL training and therefore, adaptive context-aware strategies that account for changing operational conditions are needed.

The research presented in this article aims to address the above challenges by considering the following questions.

1) *RQ1:* What techniques can be adopted to accelerate FL in an IoT environment?
2) *RQ2:* What techniques can be adopted to minimize the impact of computational heterogeneity of IoT devices?
3) *RQ3:* How can these techniques adapt to changes in network conditions?

This article presents `FedAdapt`, a holistic framework that mitigates the above challenges of accelerating FL, reducing the impact of computational heterogeneity and adapting to varying network bandwidth. To accelerate FL training and to address *RQ1*, `FedAdapt` is underpinned by an offloading technique in which the layers of a DNN model can be offloaded from a device to a server to alleviate the computational burden of training on the device. To address *RQ2*, `FedAdapt` incorporates a reinforcement learning (RL) strategy to automate the identification of layers that are offloaded from a device to the server. To address *RQ3*, `FedAdapt` further optimizes the RL strategy to develop different offloading strategies for each device while accounting for changing network bandwidth. A clustering technique is used to rapidly generate the offloading strategy.

The above are developed on a testbed comprising five devices and a server for two DNN models, namely, VGG-5 and VGG-8 [18]. The experimental studies highlight that `FedAdapt` introduces a negligible overhead (time for executing `FedAdapt` modules). The key results are that `FedAdapt` reduces the total training time of VGG-5 by 30% while achieving the same accuracy and convergence speed when compared to classic FL. Additionally, `FedAdapt` reduces the total training time of VGG-8 by 40% when compared to FL without requiring further RL training.

The *research contributions* of this article are as follows.

1) The *development of an adaptive offloading technique* that generates optimal offloading strategies for devices to reduce the impact of computational heterogeneity. The training time on the experimental test bed was observed to reduce by 40% for each round of FL using VGG-5. This is achieved by the first ever introduction of a variable layerwise training strategy to FL, which relies on offloading by using RL. To the best of our knowledge, `FedAdapt` is the first work to introduce dynamic offloading strategies into FL training.
2) The *development of* `FedAdapt`, a holistic framework that incorporates techniques for accelerating FL training, reducing the impact of computational heterogeneity, and adapting to varying network bandwidth. `FedAdapt` outperforms classic FL by reducing the training time by up to 40%.

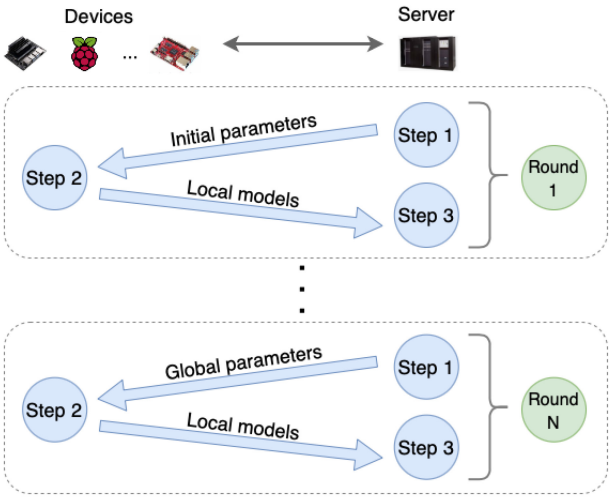


Fig. 1. Steps in each round of FL training: step 1—server initializes the parameters of the global model and sends to each device, step 2—each device completes training on its local data set and sends local model to server, and step 3—server aggregates local models to generate a new global model.

The remainder of this article is organized as follows. Section II presents the background and related work. Section III presents the `FedAdapt` framework and the problem model. Section IV presents the training process of an RL agent essential to `FedAdapt`. Section V highlights the results obtained from an experimental study. Section VI concludes this article and presents future work.

II. Background and Related Work

Federated Learning: FL [19] was developed to train ML models in a distributed manner. Each round of FL comprises three key steps as shown in Fig. 1. In the first step, a global model is initialized on the server and distributed to all devices. Each device independently trains the ML models using data generated by the device. Typically, one epoch of local training utilizes the entire data set from each device. After independently training, in the second step, the trained models from the local devices (updated model parameters) are sent to the server. In the third step, a new global model is aggregated using methods such as federated averaging (FedAvg) on the server [19]. In subsequent rounds of FL training, the aggregated model is distributed to all devices and the above steps are repeated until the training loss converges or a time limit is exceeded. FL is scalable on each device since local training can be carried out independently. However, FL is known to be less efficient for heterogeneous devices that have different computational capabilities. Thus, local training in the straggler becomes a bottleneck [8], [9].

Split Learning: SL [20] was developed to partition a monolithic DNN into two networks, namely, a device-side and a server-side network. On the device side, the DNN is trained up to the layer at which the DNN is partitioned. Then, the activation feature map of the last layer on the device is sent to the server. The server continues training until the last layer of the

TABLE I
COMPARISON OF FL, SL, SFL, AND FEDADAPT

	FL	SL	SFL	FedAdapt
Independent (parallel) training	✓	✗	✓	✓
Limited computational resources	✗	✓	✓	✓
Heterogeneous devices	✗	✗	✗	✓
Changing network bandwidth	✗	✗	✗	✓
Optimizing offloading strategy	✗	✗	✗	✓

DNN. After the training loss is calculated and the gradient is updated, the respective gradients are sent to the device so that the gradients on the device side can be calculated and updated. When training in SL with multiple devices, the devices are trained in a sequential round robin fashion whereby only one device will be connected to the server at a time. After a given device completes training, the updated weights are copied onto the next device to continue training. By training with fewer layers on the device side, computation can be significantly reduced on the device compared to FL in which the entire DNN is trained on the device. While SL is beneficial for collaborative training when there are a small number of devices, it is inefficient for a large number of participating devices due to sequential training across the devices.

Splitfed Learning: The synergy of FL and SL has been explored recently to mitigate the above limitations. Splitfed learning (SFL) was proposed to achieve both parallel training of FL and acceleration of device training in SL [21]. The combination of SFL and transfer learning has been proposed to further improve the convergence rate of large models (e.g., ResNet56) on limited resources [22]. A local loss is incorporated in the device side to reduce the communication overhead [23]. However, existing SFL-based research does not consider optimal partitioning strategies or require hardware configuration data to manually determine the model partitions for all devices before training. In addition, a static partition strategy could become suboptimal when operational conditions change during training.

Computation Offloading in Edge Computing: Computation offloading has been widely adopted in the literature of edge computing. Computationally expensive components of a distributed application that need to be executed on resource constrained devices are offloaded to a server located nearby, thereby alleviating the computational burden on the device [24]. Research on optimizing the offloading strategy to maximize performance (e.g., training time) while minimizing energy consumption has been explored [25]. However, the application of computation offloading to ML tasks on edge devices is still in the initial stages of exploration, particularly within the context of FL training. There is research on determining optimal offloading for inference [26], [27]. However, since FL training is computationally intensive and requires more time than an inference query, a more adaptable and dynamic offloading strategy that reacts to changes in operational conditions is required.

How Does FedAdapt Differ From Prior Work? Table I presents a comparison of FL, SL, SFL, and FedAdapt. As in FL, FedAdapt independently trains on the local device and, as in SL, accounts for the limited computational resources on the device. Although DNN layer offloading is also utilized in SFL, the key differences are that FedAdapt accounts for heterogeneous devices and are changing network bandwidth that affect training performance not considered within classic SFL. In addition, FedAdapt requires no prior knowledge of the devices, but uses an automated approach based on RL to identify the DNN partitions for each device, thus mitigating the challenge of heterogeneity. We also note from the literature that most FL, SL, and SFL implementations are simulation based and do not focus on real test beds. The benefits of FedAdapt on the other hand are demonstrated on a physical lab-based test bed.

TABLE II
NOTATION USED IN FEDADAPT

Notation	Description
K	Number of devices in a FL task
t	Time step at round t
k	Denote a device participating in FL
W^k	Training workload of device k
Net_t^k	Network speed at round t of device k
C_t^k	Training speed at round t of device k
$L(\mu_t^k)$	Total amount of communication of device k at round t
s	Server coordinating the FL task
C_t^s	Training speed at round t of server s
μ_t^k	Offloading strategy at round t of device k
μ_t	Offloading strategy at round t for all devices or groups
T_t^k	Local training time at round t of device k
B^k	Training time of device k without offloading
T_t	FL training time at t
S_t	State at round t in RL
A_t	Action generated by RL at round t
R_t	Reward at round t
G	Total number of groups
g	A representative device in group g
W^g	Training workload of the representative device of g
Net_t^g	Network bandwidth of the representative device of g at round t
C_t^g	Training speed of the representative device of g at round t
μ_t^g	Offloading strategy of the representative device of g at round t
T_t^g	Training time of the representative device of g at round t
f_{norm}	Normalization function used in RL to calculate R_t

III. FEDADAPT FRAMEWORK

This section provides an overview of the FedAdapt framework and the underpinning techniques. Then, the problem of distributing the DNN model in FL across the device and server for performance efficiency is formulated. Table II shows the mathematical notation we have used in this article.

A. Overview

FedAdapt (Fig. 2) comprises four modules, namely: 1) preprocessor; 2) clustering module; 3) trained RL agent; and 4) postprocessor.

After an FL round has been completed (Round $t-1$), the *preprocessor* gathers observation on the state of the devices, such as computational capabilities[1] and network bandwidth

[1]In this article, we define computational capabilities as the training time per iteration for one batch of training samples.

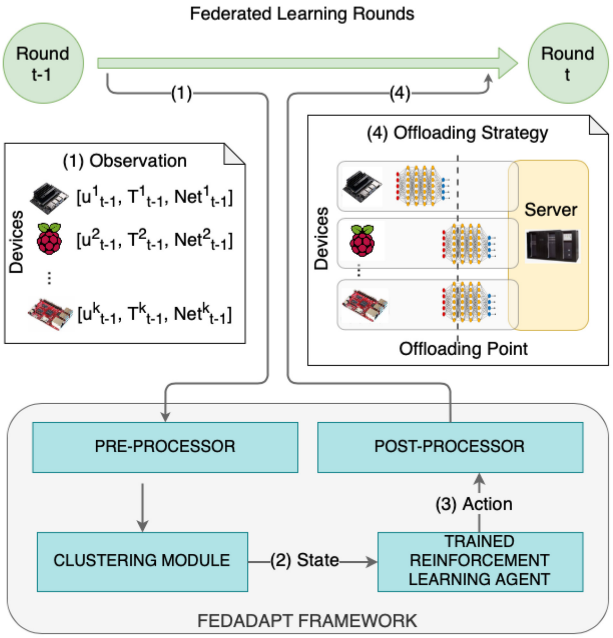


Fig. 2. FedAdapt framework and its positioning within FL.

between each device and server. The training time per iteration is normalized by the preprocessor.

The *clustering module* groups devices with similar training time into a single group. In other words, all devices within a group are considered as computationally homogeneous. A group is further defined by accounting for network bandwidth between the device and the server. RL is employed to determine the offloading strategy for each group.

The *trained RL agent* given the group information and observations (referred to as state) will generate an offloading decision (referred to as action) for each group by using a fully connected neural network. The training process of the RL agent is further discussed in Section IV.

The *postprocessor* makes use of the output of the trained RL agent and maps the offloading decision for each group on to the devices in the group. All devices in a group execute the same offloading strategy. The offloading strategy indicates which layers of the DNN model will be on each device for the FL round t.

FedAdapt is underpinned by three techniques. The first technique is *offloading* in which the DNN model used in FL is partitioned so that certain layers of a neural network can be offloaded from computationally constrained devices on to the server. The offloading approach is used to accelerate FL training since the computational workload is transferred to more capable resources that may be available on the server; this addresses RQ1 posed in Section I. The performance gain obtained by offloading will be experimentally shown in the next section.

In the offloading approach, the layer after which the DNN model is partitioned is referred to as the offloading point (OP). The OP is identified by the RL agent. The initial layers of the DNN remain on the device whereas the layers after the OP are offloaded to the server. During training, the intermediate activation and corresponding labels and gradients of the distributed DNN are exchanged between the devices and server. Although there are communication overheads in transferring the activation and gradients during training, the overall FL training time is reduced due to the gain by computational offloading (refer Section V).

The second technique uses *RL* to address the challenge of computational heterogeneity of devices that leads to stragglers in FL as posed in RQ2. This is an automated technique that enables the postprocessor to identify the OP for each individual device before a round of FL so that an optimized offloading strategy is executed for each device participating in an FL training round. To address the challenges in scaling training for a large number of devices and in determining an offloading strategy for all devices, a *clustering-based approach* is employed to group devices that have similar computational performance. Once the offloading decision for each group is determined, the postprocessor maps the decision on to each device so that the offloading strategy is executed for FL training.

The third technique employed in FedAdapt is *optimized RL* so that operational conditions, namely, network bandwidth between devices and the server, can be accounted for generating optimal offloading strategies. Thus, RQ3 raised initially is addressed by FedAdapt.

B. Problem Model

FedAdapt assumes that the network bandwidth between the device and the server can change between different FL rounds. The network bandwidth from the previous FL round is observed for generating an offloading strategy. However, any changes to the network bandwidth during a round are not accounted for. The goal is to reduce the overall training time by achieving suitable offloading strategies for all devices and adapting to network changes that are observed.

Assume that FL training is carried out with K devices, each device has a training workload W^k for each round, an FL task involving a server s has training speed C_t^s at round t, a set of participating devices $\{k\}_{k=1}^K$ has training speed C_t^k, and the network bandwidth between the device and the server Net_t^k. The offloading strategy for the device is μ_t^k denoted as the remaining proportion of computation on each device at round t. For a round of FL training on device k, the proportion of the workload that is executed on the device is $\mu_t^k W^k$ and $(1-\mu_t^k)W^k$ is offloaded to the server. Let $L(\mu_t^k)$ be the size of the feature maps that are transferred between the device and server during the training of round t. It is worth noting that $L(\mu_t^k)$ depends on μ_t^k as the offloading strategy determines the size of the transferred feature map. Finally, the training time for device k of round t may be calculated as follows:

$$T_t^k = \frac{\mu_t^k W^k}{C_t^k} + \frac{(1-\mu_t^k)W^k}{C_t^s} + \frac{L(\mu_t^k)}{Net_t^k} \tag{1}$$

where $[(\mu_t^k W^k)/C_t^k]$ and $([(1-\mu_t^k)W^k]/C_t^s)$ are the training time on the device and on the server, respectively. $([L(\mu_t^k)]/Net_t^k)$ is the communication time during training.

In round t, W^k, C_t^s, C_t^k, and Net_t^k are either constants or are variables that are not controlled by FedAdapt. The offloading strategy for each device μ_t^k is controlled by FedAdapt. In

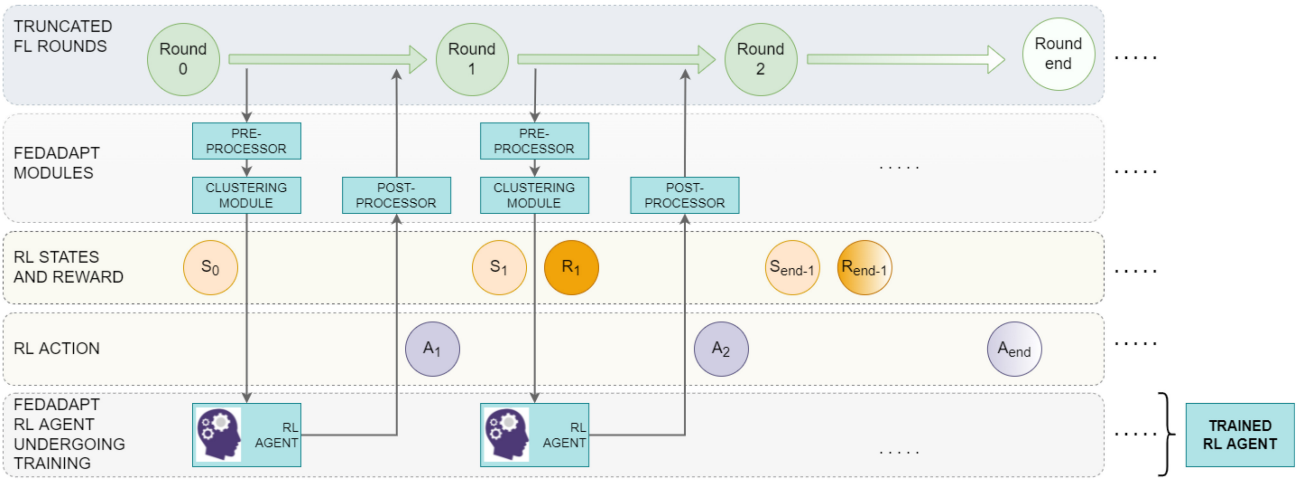


Fig. 3. Training of the RL agent used in FedAdapt.

TABLE III
COMPUTATIONAL WORKLOAD (ON THE DEVICE) AND TRAINING TIME FOR ONE ROUND

Methods	Computation	Training Time
FL	$\sum_{k=1}^{K} W^k$	$\max\{\{\frac{W^k}{C_t^k}\}_{k=1}^{K}\}$
SL	$\sum_{k=1}^{K} \overline{\mu} W^k$	$\sum_{k=1}^{K} \frac{\overline{\mu}W^k}{C_t^k} + \frac{(1-\overline{\mu})W^k}{C_t^s} + \frac{L(\overline{\mu})}{Net_t^k}$
SFL	$\sum_{k=1}^{K} \overline{\mu} W^k$	$\max\{\{\frac{\overline{\mu}W^k}{C_t^k} + \frac{(1-\overline{\mu})W^k}{C_t^s} + \frac{L(\overline{\mu})}{Net_t^k}\}_{k=1}^{K}\}$
FedAdapt	$\sum_{k=1}^{K} \mu_t^k W^k$	$\max\{\{\frac{\mu_t^k W^k}{C_t^k} + \frac{(1-\mu_t^k)W^k}{C_t^s} + \frac{L(\mu_t^k)}{Net_t^k}\}_{k=1}^{K}\}$

this article, μ_t^k is an OP. The collection of OPs for K devices is μ_t, which is $\{\mu_t^k\}_{k=1}^K$. In terms of SL and SFL, μ_t^k is uniform among all devices for all rounds, denoted as $\overline{\mu}$. In synchronous training, the server waits for all devices to complete training. The FL training time of round t is $T_t = \max\{\{T_t^k\}_{k=1}^K\}$. Table III summarizes the computational workload of K devices and the training time required for one round for different methods.

To reduce the training time for all devices in a round, we define our optimization target as minimizing the average training time for K devices as follows:

$$\begin{aligned} &\underset{\mu_t^k}{\text{minimize}} \quad \frac{1}{K}\sum_{k=1}^{K} T_t^k \\ &\text{subject to} \quad T_t^k = \frac{\mu_t^k W^k}{C_t^k} + \frac{(1-\mu_t^k)W^k}{C_t^s} + \frac{L(\mu_t^k)}{Net_t^k}. \end{aligned} \tag{2}$$

The training time of round t is T_t, which is bound by the maximum training time of all participating devices. However, FedAdapt not only optimizes the maximum training time, which may be bound by the straggler devices, but also aims at reducing the training time of each device. Reducing the training time on individual devices implies reducing the amount of computation carried out on the devices. Therefore, in FedAdapt, we define the objective as average training time for K devices. The objective of reducing the total training time over all FL rounds is achieved by lowering the average training time of each round for which μ_t is optimized for each round based on variable operational conditions, namely, C_t^s, C_t^k, and Net_t^k.

IV. TRAINING THE REINFORCEMENT LEARNING AGENT FOR FEDADAPT

In this section, the training process of the RL Agent that is employed in FedAdapt to achieve the objectives of (2) combined with a clustering technique is presented. For training the RL agent, the input state, output action, and the reward function, which are essential components of the RL technique, are presented.

RL is a sequential decision optimization technique used in a variety of domains [28]–[30], including optimization problems requiring automated control [31]. An RL-based agent is used in FedAdapt for two reasons.

1) RL provides an automated mechanism to generate reasonable offloading strategies for the participating devices, which maximizes the reward, i.e., training time in FedAdapt. Given the computational heterogeneity of IoT devices, existing research assumes that the hardware configuration of all devices can be obtained (white box), for identifying the stragglers [11], [15]. However, obtaining the hardware configuration of all devices may not be possible in a real FL application. In addition, the one-time estimation of training time of a device is also often inaccurate since certain factors, such as resource availability and network bandwidth, may change during training. RL would eliminate the need for explicitly profiling the hardware on the device by using the training record from the last round.
2) A trained RL agent can be reused for similar FL tasks. We verify the performance of reusing the RL agent without retraining in Section V-D. Without training for another specific model, the RL agent can achieve 57% training time reduction per round.

Basic Training Approach: The training approach of the RL agent is shown in Fig. 3. The state is obtained from the clustering module and comprises normalized values (for training time and action) in contrast to the observation shown in

Fig. 2. The RL agent employs a neural network with three layers that obtains the current input state (S_t) as input. The RL agent produces the offloading action A_t, which is different to the offloading strategy produced by the postprocessor in that the action is a value between 0 and 1 for a device group, but the offloading strategy is a mapping of this value on to an OP for each device. The trained RL agent is obtained at the end of training. The aim of the RL agent is to maximize the accumulated rewards over each round, which is in line with (2). The training process begins after the first round with classic FL training (no offloading) used to generate the initial state S_0.

Clustering Technique With RL: A naive design strategy would be to generate an offloading action for each device. However, this strategy would be limited in the following two ways. First, if the number of participating devices changes during FL training, the RL agent will fail to generate an offloading action due to the fixed input and output dimensions of the neural network used by the RL agent at the beginning of FL training. Second, when the number of devices K becomes large, it is challenging to train the RL agent due to the large action space that will need to be explored. The action space grows exponentially with the increase in the number of devices (for example, consider K devices and a DNN model with N layers, then the size of the action space is N^K).

Therefore, a clustering technique is utilized at the beginning of each FL round (except S_0). In the clustering process, homogeneous devices are first grouped according to the training time per iteration and network bandwidth between the device and server into G groups (the no. of groups is determined by heuristic algorithms, such as the elbow method [32]). Then, G groups are used instead of K devices for the input state and output actions dimension. Therefore, the objective is formulated as

$$\begin{aligned} &\underset{\mu_t^g}{\text{minimize}} \quad \frac{1}{G}\sum_{g=1}^{G} T_t^g \\ &\text{subject to} \quad T_t^g = \frac{\mu_t^g W^g}{C_t^g} + \frac{\left(1-\mu_t^g\right)W^g}{C_t^s} + \frac{L\left(\mu_t^g\right)}{Net_t^g} \end{aligned} \tag{3}$$

where g is defined as a representative device in the group that has the maximum training time. In other words, for each group, `FedAdapt` treats all devices in a group as homogeneous devices, which means that they have similar computation capability and network bandwidth. In `FedAdapt`, the device with the maximum length of training time is used to represent the group. Therefore, W^g, Net_t^g, C_t^g, μ_t^g, and T_t^g are bounded by the representative device in each group.

Optimizing for Network Bandwidth: The offloading strategy will need to change when the operational condition, namely, network bandwidth between the device and the server changes (the change in OP when network bandwidth changes will be demonstrated in Section V). This is to optimize the performance of `FedAdapt`. The RL method will need to generate a different output action when the network bandwidth changes. An intuitive approach is to train the RL agent for different network bandwidths. However, in practice, we observed that this provides suboptimal offloading actions due to the rewards that are dominated by when the network bandwidth is not limited. To circumvent this, in `FedAdapt`, devices with limited network bandwidth are considered within an additional heterogeneous group and devices are dynamically added to this group. At the beginning of each FL round, the network bandwidths of all devices are observed. If the network bandwidth drops below a threshold (discussed in Section V) for a device, then it is assigned to the additional group. The training of the RL agent is carried out in a controlled environment such that the network bandwidth between the device and the server is limited to represent the group.

State and Action: The maximum local training time of the device in a group is used in the input state. The RL agent in each training round will produce the offloading action for each group. The action of a group is mapped by the postprocessor to the DNNs of all devices in the group, which is μ_t^g. For instance, a VGG-5 [18] model with three convolutional layers and two fully connected layers has five offloading actions. The output action for each group is designed to be a real value (μ_t^g) ranging from zero to one so that the RL agent adapts to multiple DNN models. This is mapped to the percentage of the total computational workload of the DNN that is placed on the device. After obtaining μ_t^g, the number of floating point operations (FLOPs) is calculated and set as the target workload on devices. The OP closest to the target workload is chosen. Equation (4) shows the input state and output action at round t

$$\begin{aligned} S_t &= \left\{T_t^g, \mu_{t-1}^g\right\}_{g=1}^{G} \\ A_t &= \left\{\mu_t^g\right\}_{g=1}^{G} \\ &\text{subject to} \quad \mu_t^g \in (0,1]. \end{aligned} \tag{4}$$

Reward Function: The reward function guides the training process of the RL agent. The reward obtained at the end of each FL training round is denoted as R_t. To achieve the objective of (2), one option is to set the reward as the average training time. However, in practice, the device with the largest training time will dominate the reward. Hence, a normalization function (f_{norm}) is used to calculate the reward. The training time for each device when no DNN model is offloaded is denoted as B^k (a baseline). The training time of device k (T_t^k) is normalized with B^k using (5) as follows:

$$\begin{aligned} R_t &= \sum_{k=1}^{K} f_{\text{norm}}\left(T_t^k, B^k\right) \\ f_{\text{norm}} &= \begin{cases} 1-\frac{T_t^k}{B^k}, & T_t^k \le B^k \\ \frac{B^k}{T_t^k}-1, & T_t^k > B^k. \end{cases} \end{aligned} \tag{5}$$

Choice of Algorithm: A variety of algorithms is available to train the RL agent for achieving the objectives presented in Section III-B; examples include DQN [33], [34] and REINFORCE [35]. In this research, the proximal policy optimization (PPO) [36] is chosen. It is a state-of-the-art method, which is relatively easy to use and has good performance on standard RL benchmarks [36]. Furthermore, compared to DQN that determines the optimal action by evaluating all the possible actions using the Q-network [34], PPO

generates the output as an explicit action by using a policy network [36]. If DQN is adopted for determining the offloading action for each group during FL training, all possible OPs for each group will need to be evaluated using the Q-network [34]. However, this is not possible in FedAdapt since the action space is continuous ($\mu_t^g \in (0, 1]$), thereby making it impossible to enumerate all actions. Compared to on-policy algorithms, such as REINFORCE, PPO is an off-policy RL algorithm, which repeatedly uses the trajectory data from previous explorations (interactions between the agent and the environment). This improves the training efficiency given that exploration is time consuming. Thus, we choose PPO for the RL algorithm.

RL Training Methodology: The RL agent comprises two fully connected networks, namely, the actor and critic networks. Both networks have the same architecture comprising three layers. When the RL agent is trained, the critic network is adopted for assisting the training of the actor network. The actor network is trained to output the offloading action. After completing training, only the actor network will be used to provide the offloading action. Ideally, the RL agent should be trained online during an FL task. However, if the RL agent is trained online during an FL task, the learning time is the time for all rounds in FL training. The RL agent will need to wait until the completion of each round to obtain the training time required for calculating the reward. Therefore, we train the RL agent in an offline manner before FL tasks. To accelerate the training of the RL agent, the number of batches used for each round is reduced, referred to as truncated FL rounds in Fig. 3. In addition, we collect training time per batch for each device instead of the training time of a round as an element of the input state and output action. The FL model will be trained again with normal rounds (beyond the truncated rounds) after the trained RL agent is obtained.

V. Experimental Studies

In this section, the performance of FedAdapt is experimentally verified. The section is organized in response to the research questions raised in Section I. Section V-A presents the results obtained from examining DNN layer offloading in FL (addresses RQ1 on accelerating FL training). Section V-B highlights the results of the RL technique used in FedAdapt (addresses RQ2 on minimizing the impact of computational heterogeneity of devices). Section V-C presents the results when the RL technique is optimized to account for changing network bandwidth (addresses RQ3). The benefits of FedAdapt are demonstrated by comparing with classic FL.

A. Layer Offloading in FL

The assumption that layer offloading can accelerate FL training on computationally limited IoT devices (for example, single board computers, such as Raspberry Pi) is verified. An empirical study is carried out under different network bandwidth values using two CNNs, namely, VGG-5 [18] and VGG-8 [18]. The network bandwidth values correspond to WiFi (75 and 50 Mb/s; same uplink and downlink bandwidth), 4G+ (25-Mb/s uplink and 50-Mb/s downlink), and 4G (10-Mb/s uplink and 20-Mb/s downlink) connections. The study examines the performance of FL for all OPs of the CNNs with different network bandwidth.

TABLE IV
Architecture of the Models Used for Evaluating FedAdapt. Convolution Layers Are Denoted by C Followed by the Number of Filters, Filter Size Is 3 × 3 for All Convolution Layers, MaxPooling Layer Is MP, Fully Connected Layer Is FC With a Given Number of Neurons, and OP Is OP With Index

Model	Architecture
VGG-5	C32-MP(OP1)-C64-MP(OP2)-C64(OP3)-FC128-FC10(OP4)
VGG-8	C32-C32-MP(OP1)-C64-C64-MP(OP2)-C128-C128(OP3)-FC128-FC10(OP4)

TABLE V
Training Time Per Iteration When Layer Offloading Is Used in FL for VGG-5 Under Different Network Bandwidth

OP	75Mbps (Wi-Fi)	50Mbps	25Mbps (4G+)	10Mbps (4G)
OP1	**2.38**	**2.7**	**3.52**	6.07
OP2	3.61	3.9	4.36	5.31
OP3	5.24	5.26	5.42	6.73
OP4 (device native)	4.36	4.36	4.36	**4.36**

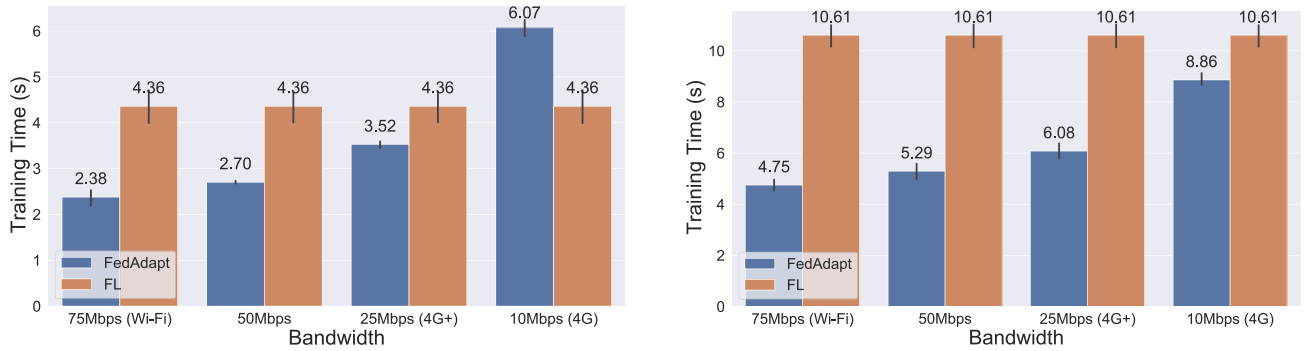
The study will demonstrate that: 1) layer offloading from a device to a server reduces the FL training time compared to classic FL in which all layers of the DNN execute on the device. Previous studies highlight that computational time on devices is a major bottleneck for resource constrained devices in FL [8], [9], [22] and 2) the performance gain (reduction in training time) is substantial and offsets the communication overhead that is incurred in transferring the activation and gradient feature maps between the device and the server.

Setup: The testbed includes an edge server with a 2.5-GHz dual-core Intel i7 CPU and an IoT device, namely a Raspberry Pi 4 Model B with 1.5 GHz quad-core ARM Cortex-A72 CPU. Only a single device is used to validate the performance gain of FL training with layer offloading. Wi-Fi for the device is supported by the Virgin Media Super Hub 3 router.

The Linux built-in network traffic control module tc is employed to emulate different bandwidths between the device and the server. The standard representation of VGG-5 and VGG-8 models used in this study are shown in Table IV. For simplicity the batch normalization and nonlinear layer (ReLU) are not shown. The layers denoted with OP present the OPs empirically tested in this study (all layers after the OP can be offloaded to the server). The CIFAR-10 data set is used and a batch size of 100 is used for all experiments.

Results: Tables V and VI present the training time per iteration of FL when using layer offloading for all OPs of VGG-5 and VGG-8. VGG-5 and VGG-8 both have four OPs (since the FLOPs of the last dense layer is small, an OP is not considered between two dense layers). The last OP in each model (OP4) corresponds to device native execution of the DNN as in classic FL. The results are an average of five independent runs. The best result for each value of network bandwidth is in bold.

The best values of training time per iteration for VGG-5 and VGG-8 are 2.38 and 4.75 s compared to 4.36 and 10.61 s for classic FL in a Wi-Fi network. The best OP is OP1 both



(a) (b)

Fig. 4. Comparing training time per iteration in FedAdapt and classic FL. (a) VGG-5. (b) VGG-8.

TABLE VI
TRAINING TIME PER ITERATION WHEN LAYER OFFLOADING IS USED IN FL FOR VGG-8 UNDER DIFFERENT NETWORK BANDWIDTH

OP	75Mbps (Wi-Fi)	50Mbps	25Mbps (4G+)	10Mbps (4G)
OP1	**4.75**	**5.29**	**6.08**	**8.84**
OP2	7.52	8.37	8.32	9.95
OP3	10.74	11.98	12	15.93
OP4 (device native)	10.61	10.61	10.61	10.61

for VGG-5 and VGG-8. Performance is improved when the majority of layers are offloaded from the resource constrained device onto the server. The training time is thus reduced by over 45% for VGG-5 and over 55% for VGG-8 compared to classic FL.

The best OP for both models is the pooling layer—all layers beyond the pooling layer are offloaded. This is because pooling is a relatively low computationally intensive workload and has a low volume of data (activation feature maps) that needs to be transferred between the device and the server.

The best performance achieved for the four different network bandwidths is reported in Fig. 4(a) and (b) for the VGG-5 and VGG-8 model, respectively. More than 45% and 55% of the training time per iteration can be reduced for the Wi-Fi connection on the VGG-5 and VGG-8, respectively. With the decrease of network bandwidth, the performance acceleration is reduced. For 25-Mb/s network bandwidth, which is the typical bandwidth of real time 4G+ mobile network,[2] the training time is reduced by 19% and 43% on VGG-5 and VGG-8, respectively. When the bandwidth is lowered to 10 Mb/s, offloading has negative effect on the VGG-5 model and the best OP is the OP4 (device native training). However, for the VGG-8 model, offloading reduces 17% of the training time at 10-Mb/s bandwidth.

Offloading is influenced by the network bandwidth between the device and the server since there is frequent communication during training. In the case of VGG-5, device native execution is performance efficient for a bandwidth of 10 Mb/s. However, offloading the layers after OP1 from the device to the server is more effective for higher bandwidths. In the case of VGG-8, offloading is more effective than device native execution. Therefore, FedAdapt is envisioned to be beneficial in scenarios where distributed IoT devices have limited computational resources and need to offload the FL training workload onto a server. Examples include home security cameras that use Wi-Fi and leverage computational resources on a home hub [9], [37], [38] and wearable visual auxiliary equipment that operates in both indoor and outdoor environments (Wi-Fi and on mobile networks) [39]–[41]. When the network changes as a device moves across the coverage offered by different networks, then FedAdapt appropriately selects between device native and offloading-based FL.

[2]https://www.4g.co.uk/how-fast-is-4g/

B. RL Optimization for Heterogeneity

Computational heterogeneity of devices leads to the challenge of a straggler (a computationally weak device prolonging the training time of each round) in FL. This is because resource availability of an IoT device will vary due to varying hardware architectures and given that other applications running on the device may require more resources. A straggler device can negatively impact the overall training time. Therefore, an offloading strategy that can account for device heterogeneity and minimize the impact of the straggler is required.

An RL agent that can select different OPs for the devices in FL is designed. The training process of the RL agent is guided by the reward function shown in (5). In this experiment, the focus is on device heterogeneity and therefore changing network bandwidth is not considered.

Setup: The IoT-edge server environment considered has one server and five devices. The server is the same as used in Section V-A. The devices are: 1) two Raspberry Pi 4 (denoted as $Pi4^1$ and $Pi4^2$) presented in Section V-A; 2) two Raspberry Pi 3 (denoted as $Pi3^1$ and $Pi3^2$) Model B with 1.2-GHz quad-core ARM Cortex-A53 CPU; and 3) one Jetson Xavier NX (denoted as Jetson) with embedded GPUs. The running CPU frequency of $Pi4^2$ is set to 0.7 GHz to create a straggler in a controlled manner.

All Raspberry Pis have the same version of the Raspbian GNU/Linux 10 (Buster) operating system, Python version 3.7, and PyTorch version 1.4.0. The Jetson and the server have the same version of Python and PyTorch. CuDNN library is installed on the Jetson in order to use the GPU during training. All devices are connected to the server in a Wi-Fi network

using a router (presented in Section V-A). The average available bandwidth between the device and server is 75 Mb/s. The experiments are carried out in a real-world environment with five IoT devices, which is similar to testbeds employed in peer-reviewed research on FL [8], [9], [42].

The DNN model used is VGG-5. It will be demonstrated in Section V-D that the RL agent trained for VGG-5 can also be employed for VGG-8. CIFAR-10 [43] is used as the training and testing data set that contains 50K training and 10K testing samples. The training samples are uniformly divided for the five devices without overlapping samples. The entire test data set is available on the server. The number of FL rounds is 100 and the standard FedAvg [19] aggregation method is used in the server. The horizontal flip technique is used for data augmentation with a probability of 0.5 and the stochastic gradient descent (SGD) is utilized as the optimizer for updating the model parameters. The learning rate is 0.01 at the start of the FL task and 0.001 at the start of the 50th round.

Training the RL Agent: In the experiments reported in this section, the RL agent is first trained and then deployed as a trained agent to the FL task for generating offloading strategies during each round. The number of iterations in one round of FL is reduced from 100 to 5 iterations when the RL agent is trained as presented in Section IV. The RL agent has the same training schedule of 50 rounds. PPO is used as the RL algorithm. The RL agent has an actor network and a critic network. Both networks have the same architecture of fully connected layers with two hidden layers (64 and 32 neurons, respectively). The actor network is used to generate the offloading actions whereas the critic network is used by the RL algorithm to evaluate the value of a given state. During training, a discount factor $\gamma = 0.9$ is set for the RL agent to determine the importance of using reward from future states and the learning rates for the actor and critic networks are configured to be 1e−4 (these are standard values used in RL training). The actor and critic networks are updated every ten rounds, and during each update the data collected in the previous ten rounds are used 50 times. The standard deviation of the actor network is set as 0.5 at the beginning of the RL training and exponentially decayed (decay rate 0.9) after 200 rounds of training. This is to ensure that in the first 200 rounds, the RL agent has more freedom to explore the action space, but is then decreased to ensure that the RL agent will produce actions that can generate offloading strategies that will reduce the training time. These hyperparameters remain the same throughout the experiments for better generalization.

Clustering: The initial state S_0 is executed without any offloading. Table VII shows the results of clustering the five devices in the testbed. The Jetson has the faster training speed due to GPU acceleration. Pi4^2 is the straggler due to the lower CPU frequency (0.7 GHz). Using the values of the training time per iteration of each device, the k-means clustering algorithm is used to divide all devices into G groups based on the results from the first round of training. The device training time in the first round is used to cluster the devices into groups. It is assumed that the training speed of devices will not change substantially in subsequent rounds. In this experiment, $G = 3$. The Jetson and Pi4^2 (0.7 GHz) are individually allocated to a group, whereas Pi4^1, Pi3^1, and Pi3^2 are clustered into one group. The RL agent will generate the offloading actions for each group in each round.

TABLE VII
CLUSTERING DEVICES INTO GROUPS WHEN USING VGG-5

Devices	Training time (s)	Group no.	Group center
Jetson	0.07	**1**	0.07
Pi4^1 1.5GHz	3.58	**2**	3.7
Pi3^1 1.2GHz	3.75	**2**	3.7
Pi3^1 1.2GHz	3.77	**2**	3.7
Pi4^2 0.7GHz	5.14	**3**	5.14

TABLE VIII
TRAINING TIME PER ITERATION IN SECONDS FOR EACH DEVICE FOR ALL POSSIBLE OPS IN VGG-5

OP	Jetson	Pi4^1 1.5GHz	Pi3^1 and Pi3^2 1.2GHz	Pi4^2 0.7GHz
OP1	0.51	**2.38**	**2.99**	**2.63**
OP2	0.28	3.61	3.97	4.68
OP3	0.27	5.24	4.93	5.88
OP4 (device native)	**0.17**	4.36	4.47	5.15

Results: First, all potential OPs for each device are empirically tested to generate the ground truth so as to verify the offloading actions produced by the RL agent and the eventual offloading strategies generated by FedAdapt; this is shown in Table VIII for VGG-5. The best performance result is shown in bold. All results are an average of five independent runs. The optimal offloading actions for each group is $[\mu_{G=1} > 0.96, \mu_{G=2} < 0.38, \mu_{G=3} < 0.38]$. The boundary of all OPs was tested and the borderline between OP1 and OP2 is determined as 0.38, OP2 and OP3 as 0.79, and OP3 and OP4 as 0.96. The proportion of workload (FLOPs) on the device is 0.1, 0.66, 0.94, and 1 based on the OP. The OP closest to the action generated by the RL agent is chosen. The boundaries of an OP are the mean of pairwise adjacent OPs (0.38, 0.79, and 0.96). Best performance is obtained for all Raspberry Pis when the layers after OP1 are offloaded to the server and for the Jetson is executed device native.

The empirical results from Table VIII are used to verify the offloading actions of the RL agent for VGG-5. The action of the RL agent for 500 rounds (or 500 truncated FL rounds) for VGG-5 is shown in Fig. 5. The results are the average of five independent runs with different random seeds and are shown for three different groups, G_1, G_2, and G_3. The horizontal lines for OP1, OP2, OP3, and OP4 show the boundaries for each OP. At the start of RL training, the RL agent produces similar offloading actions (around 0.5) for each group. However, the RL agent optimizes the offloading actions for each group guided by rewards. The mean actions of G_1, G_2, and G_3 become optimal after the 80th, 30th, and 40th rounds, respectively. After the 80th round, the mean actions of all three group are in line with the optimal offloading actions.

When the training of the RL agent is complete, the trained actor network is deployed to guide the offloading strategies. The average training time for one round for each device using VGG-5 is shown in Fig. 6. The training times for all

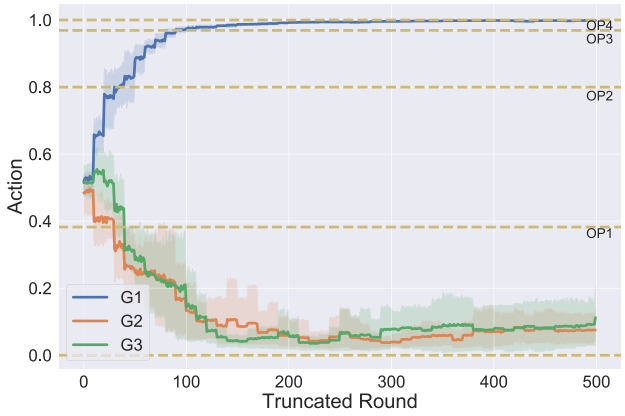


Fig. 5. Actions produced for each group chosen by the RL agent during training for VGG-5.

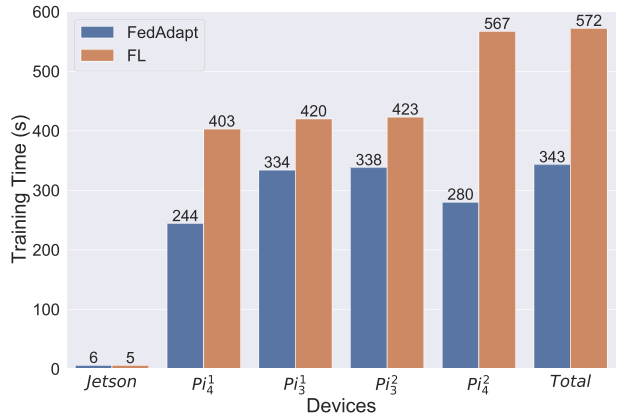


Fig. 6. Device and total training time per round in seconds in FedAdapt and classic FL using VGG-5.

Raspberry Pis are reduced and any negative impact of offloading is minimized on the Jetson. The maximum performance gain is for the straggler Pi4^2—a 50% reduction in training time per round is observed. FedAdapt saves 40% of the total training time for one round compared to classic FL. In this experiment, the network bandwidth is not changed during the FL rounds.

C. Adapting to Changing Network Bandwidth

This section evaluates FedAdapt to address RQ3. Limited network bandwidth negatively impacts performance when offloading is used (Table V). For example, if the bandwidth of the device decreases from 75 to 10 Mb/s, then the OP will need to change. The experimental setup is similar to Section V-B.

Clustering: FedAdapt employs an additional group for devices with low network bandwidth and dynamically groups devices after each round. This is similar to the clustering process presented in Section V-B, but considers the network bandwidth in addition to the training time per iteration. By considering devices with low bandwidth as a separate group any negative impact on training time is reduced. The upload bandwidth of Pi3^2 is manually set to 10 Mb/s (other devices are connected via the 75-Mb/s Wi-Fi network). Three groups are employed ($G = 3$). The Jetson, Pi4^1, Pi4^2, and Pi3^1 are clustered into two groups based on their training time. The clustering into groups is shown in Table IX.

TABLE IX
EXAMPLE OF CLUSTERING INTO GROUPS FOR FIVE DEVICES WITH ONE LOW BANDWIDTH DEVICE WHEN USING VGG-5

Device	Training time (s)	Bandwidth	Group no.	Group center (s)
Jetson	0.07	75Mbps	**1**	0.07
Pi4^1 1.5GHz	3.58	75Mbps	**2**	4.16
Pi3^1 1.2GHz	3.75	75Mbps	**2**	4.16
Pi3^2 1.2GHz	3.77	10Mbps	**3**	3.7
Pi4^2 0.7GHz	5.14	75Mbps	**2**	4.16

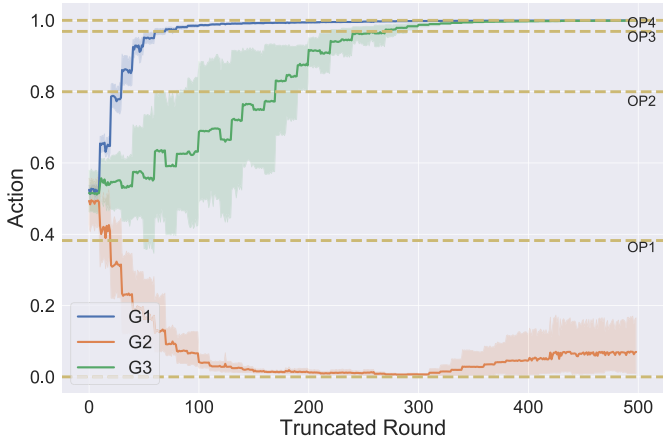


Fig. 7. Actions produced for each group chosen by the RL agent during training of VGG-5 that accounts for devices with low network bandwidth to the server.

Results: The action for each group (G_1, G_2, and G_3) for 500 rounds generated by the RL agent is shown in Fig. 7 when training VGG-5. The results are the average of five independent runs with different random seeds. The horizontal lines for OP1, OP2, OP3, and OP4 show the boundaries for each OP. At the beginning of training, the RL agent rapidly learns for G_1 and G_2. After the 20th and 60th rounds, the mean action of G_1 and G_2 becomes optimal, respectively. The optimal action for G_3 is determined by the RL agent only after the 240th round with more exploration. This is because at the beginning of training, the reward from G_1 and G_2 dominates the total reward, which guides the agent in optimizing the action. However, when offloading actions for G_1 and G_2 become optimal, the agent gradually learns for G_3. The mean actions of all three groups are in line with the optimal offloading actions after the 240th round.

D. Comparing FedAdapt and Classic FL

The performance of FedAdapt is compared with classic FL on the dimensions of training time and accuracy. The environment is set up on the five devices used previously for FL training on the CIFAR-10 data set in 100 rounds. During the first 50 rounds of FL training all devices are connected with Wi-Fi (75-Mb/s bandwidth). The remaining 50 rounds are divided into five equal time slots to lower the network bandwidth of the specific devices to 10 Mb/s in a controlled manner. The sequence is Jetson (50th to 59th round), Pi4^1

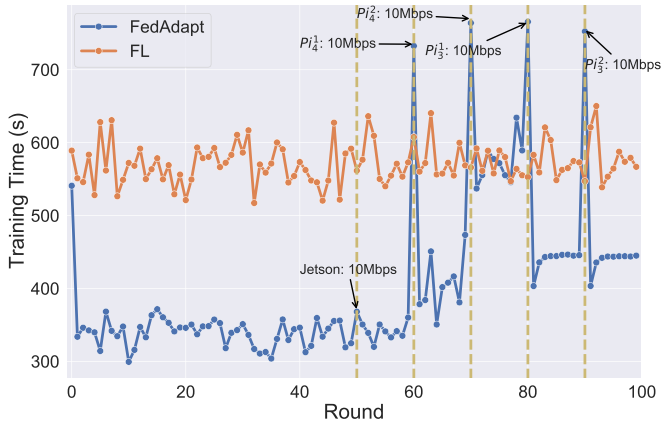


Fig. 8. Comparing the training time per round in FedAdapt and classic FL for VGG-5. Vertical lines show five time slots after the 50th round. In each slot, the highlighted device is limited to under 10-Mb/s bandwidth.

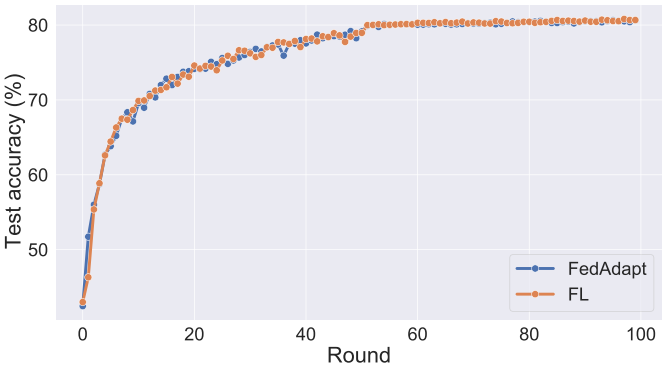


Fig. 9. Test accuracy per round of FedAdapt and classic FL for VGG-5.

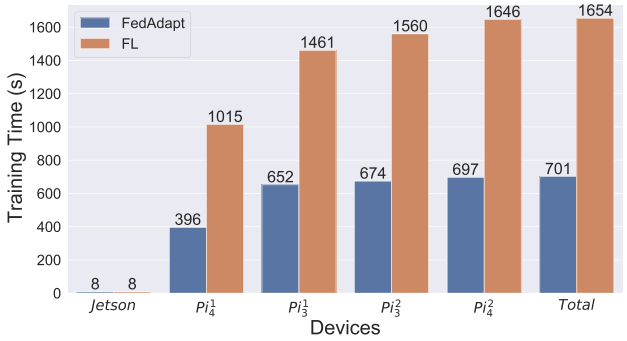


Fig. 10. Device and total training time per round in seconds in FedAdapt and classic FL for VGG-8 when using the RL agent trained for VGG-5.

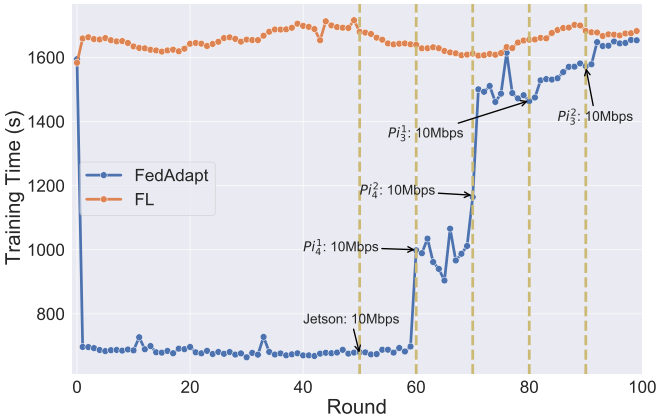


Fig. 11. Comparing the training time per round in FedAdapt and classic FL for VGG-8 when using the RL agent trained for VGG-5. Experimental setting is presented in Section V-C.

(60th to 69th round), Pi4^2 (70th to 79th round), Pi3^1 (80th to 89th round), and Pi3^2 (90th to 99th round). The FedAdapt trained RL agent from Section V-C is deployed to produce the offloading action for each device in the FL rounds using VGG-5. For classic FL, training on devices is done without offloading.

Fig. 8 shows the training time of VGG-5 for each round of FedAdapt and classic FL. Until the 50th round, FedAdapt reduces the average training time by 40% in comparison to classic FL. For the training of the last 50 rounds, the network bandwidth changes for different devices. FedAdapt responds to the changes by using observations from the previous round. Then, a suitable offloading strategy for the current round is obtained. Since the optimal action for the Jetson is OP4 (device native), there is limited impact on training time (Round 50). For the other devices, the change of bandwidth makes the offloading action invalid. However, FedAdapt adapts to these changes in the next round by reassigning the device into G_3. For the overall 100 rounds, FedAdapt reduces the training time by nearly 30% compared to classic FL.

Fig. 9 compares the test accuracy of VGG-5 using FedAdapt and classic FL for 100 rounds. Both have similar accuracy. FedAdapt employs the FedAvg algorithm as in classic FL. Therefore, FedAdapt achieves the same convergence speed and final accuracy as classic FL. The overhead incurred by FedAdapt was measured, which comprises the time for running the RL agent's actor network and the time for redeploying models on each device. An average overhead of 1.6 s was incurred (0.5% of the time for one round of training). In short, the overhead in using FedAdapt is negligible and the performance gain achieved outperforms classic FL.

Reusing the RL Agent of FedAdapt Trained for VGG-5 on VGG-8: The performance of FedAdapt using the RL agent that was trained for VGG-5 was evaluated on VGG-8. The trained RL agent on VGG-5 (presented in Section V-C) is used without custom retraining for VGG-8. The average training time of one round for each device is shown in Fig. 10. The maximum performance gain is for the straggler Pi4^2—a 57% reduction in training time per round is observed. Overall, FedAdapt saves 57% of training time compared to classic FL. Although FedAdapt reduces the training time when the network bandwidth changes, it generates suboptimal offloading strategies after the 70th round. Instead of selecting offloading-based strategies for devices that have a low bandwidth for VGG-8, the RL agent selects device native strategies, thereby minimizing the performance gain. This is because the RL agent is trained for VGG-5 in which device native strategies have maximum performance gain. Nonetheless, the overall training time is reduced by nearly 40% when compared to classic FL as shown in Fig. 11.

VI. Conclusion

Classic FL is impractical in IoT-edge environments given the limited computational capacity on IoT devices, heterogeneity of devices, and varying network bandwidth between the device and server, all of which significantly affect training performance. This article presented `FedAdapt`, a holistic framework that surmounts the above limitations by incorporating three techniques for accelerating FL, reducing the impact of stragglers and adapting to varying network bandwidth. `FedAdapt` reduces the training time of stragglers by over half compared to classic FL. When faced with stragglers and changing network bandwidth `FedAdapt` outperformed classic FL by reducing training time up to 40% while achieving the same accuracy and convergence speed with negligible execution time overhead.

Limitations and Future Work: `FedAdapt` relies on RL and clustering for generating offloading strategies for each participating device. However, as the number of devices becomes large, a single RL agent may not be suitable to generate offloading strategies for all devices. Distributed RL agents will need to be investigated along with hierarchical clustering of devices. In addition, since `FedAdapt` employs offloading-based training to accelerate training on IoT devices, additional communication overheads between the devices and the server are introduced. Techniques such as quantization may reduce the communication cost. These will be explored in future work.

References

[1] C. Perera, R. Ranjan, L. Wang, S. U. Khan, and A. Y. Zomaya, "Big data privacy in the Internet of Things era," *IT Prof.*, vol. 17, no. 3, pp. 32–39, May/Jun. 2015.

[2] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.

[3] B. Qian *et al.*, "Orchestrating the development lifecycle of machine learning-based IoT applications: A taxonomy and survey," *ACM Comput. Surveys*, vol. 53, no. 4, pp. 1–47, 2020.

[4] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 1–19, 2019.

[5] P. Kairouz *et al.*, "Advances and open problems in federated learning," 2019, *arXiv:1912.04977*.

[6] C. Briggs, Z. Fan, and P. Andras, "A review of privacy preserving federated learning for private IoT Analytics," 2020, *arXiv:2004.11794*.

[7] A. Das and T. Brunschwiler, "Privacy is what we care about: Experimental investigation of federated learning on edge devices," in *Proc. 1st Int. Workshop Challenges Artif. Intell. Mach. Learn. Internet Things*, 2019, pp. 39–42.

[8] C. Wang, Y. Yang, and P. Zhou, "Towards efficient scheduling of federated mobile devices under computational and statistical heterogeneity," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 394–410, Feb. 2021.

[9] Y. Gao *et al.*, "End-to-end evaluation of federated learning and split learning for Internet of Things," in *Proc. Int. Symp. Rel. Distrib. Syst.*, 2020, pp. 91–100.

[10] A. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[11] L. Li, H. Xiong, Z. Guo, J. Wang, and C.-Z. Xu, "SmartPC: Hierarchical pace control in real-time federated learning system," in *Proc. IEEE Real Time Syst. Symp.*, 2019, pp. 406–418.

[12] S. Dhakal, S. Prakash, Y. Yona, S. Talwar, and N. Himayat, "Coded federated learning," in *Proc. IEEE Globecom Workshops*, 2019, pp. 1–6.

[13] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous online federated learning for edge devices with non-IID data," 2019, *arXiv:1911.02134*.

[14] A. Imteaj, U. Thakker, S. Wang, J. Li, and M. H. Amini, "Federated learning for resource-constrained IoT devices: Panoramas and state-of-the-art," 2020, *arXiv:2002.10610*.

[15] Z. Xu, F. Yu, J. Xiong, and X. Chen, "Helios: Heterogeneity-aware federated learning with dynamically balanced collaboration," 2019, *arXiv:1912.01684*.

[16] I. Hakimi, S. Barkai, M. Gabel, and A. Schuster, "Taming momentum in a distributed asynchronous environment," 2019, *arXiv:1907.11612*.

[17] K. Bonawitz *et al.*, "Towards federated learning at scale: System design," 2019, *arXiv:1902.01046*.

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

[19] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Artif. Intell. Stat.*, 2017, pp. 1273–1282.

[20] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," 2018, *arXiv:1812.00564*.

[21] C. Thapa, M. A. P. Chamikara, and S. Camtepe, "Splitfed: When federated learning meets split learning," 2020, *arXiv:2004.12088*.

[22] C. He, M. Annavaram, and S. Avestimehr, "Group knowledge transfer: Federated learning of large CNNs at the edge," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1–13.

[23] D.-J. Han, H. I. Bhatti, J. Lee, and J. Moon, "Accelerating federated learning with split learning on locally generated losses," in *Proc. FL-ICML*, 2021, pp. 1–12.

[24] T. Zheng, J. Wan, J. Zhang, C. Jiang, and G. Jia, "A survey of computation offloading in edge computing," in *Proc. IEEE Int. Conf. Comput. Inf. Telecommun. Syst. (CITS)*, 2020, pp. 1–6.

[25] Z. Ali, L. Jiao, T. Baker, G. Abbas, Z. H. Abbas, and S. Khaf, "A deep learning approach for energy efficient computational offloading in mobile edge computing," *IEEE Access*, vol. 7, pp. 149623–149633, 2019.

[26] L. Lockhart, P. Harvey, P. Imai, P. Willis, and B. Varghese, "Scission: Performance-driven and context-aware cloud-edge distribution of deep neural networks," in *Proc. 13th IEEE/ACM Int. Conf. Utility Cloud Comput.*, 2020, pp. 257–268.

[27] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[28] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2017, pp. 3389–3396.

[29] A. El Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electron. Imag.*, vol. 19, pp. 70–76, Jan. 2017.

[30] A. R. Sharma and P. Kaushik, "Literature survey of statistical, deep and reinforcement learning in natural language processing," in *Proc. IEEE Int. Conf. Comput. Commun. Autom.*, 2017, pp. 350–354.

[31] H. Mao *et al.*, "Park: An open platform for learning augmented computer systems," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 2490–2502.

[32] T. M. Kodinariya and P. R. Makwana, "Review on determining number of cluster in *K*-means clustering," *Int. J.*, vol. 1, no. 6, pp. 90–95, 2013.

[33] R. S. Sutton *et al.*, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 99, 1999, pp. 1057–1063.

[34] V. Mnih *et al.*, "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.

[35] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep exploration via bootstrapped DQN," 2016, *arXiv:1602.04621*.

[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[37] T. D. Nguyen, S. Marchal, M. Miettinen, H. Fereidooni, N. Asokan, and A.-R. Sadeghi, "DÏoT: A federated self-learning anomaly detection system for IoT," in *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 756–767.

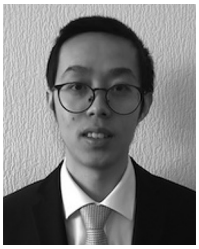
[38] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, and S. W. Kim, "A survey on resource management in IoT operating systems," *IEEE Access*, vol. 6, pp. 8459–8482, 2018.

[39] Y. Chen, X. Qin, J. Wang, C. Yu, and W. Gao, "FedHealth: A federated transfer learning framework for wearable healthcare," *IEEE Intell. Syst.*, vol. 35, no. 4, pp. 83–93, Jul./Aug. 2020.

[40] H. Guo, H. Lin, S. Zhang, and S. Li, "Image-based seat belt detection," in *Proc. IEEE Int. Conf. Veh. Electron. Safety*, 2011, pp. 161–164.

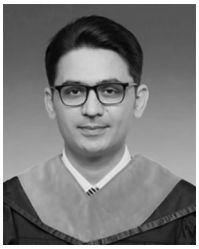
[41] W. Zhuang *et al.*, "Performance optimization of federated person re-identification via benchmark analysis," in *Proc. 28th ACM Int. Conf. Multimedia*, 2020, pp. 955–963.

[42] T. Zhang, C. He, T. Ma, L. Gao, M. Ma, and S. Avestimehr, "Federated learning for Internet of Things: A federated learning framework for on-device anomaly data detection," 2021, *arXiv:2106.07976*.
[43] A. Krizhevsky, "Learning multiple layers of features from tiny images," Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Rep. TR-2009, 2009.



Di Wu received the B.S. degree in information system and information management from Northeast Forestry University, Harbin, China, in 2015, and the M.S. degree in data science from the University of Southampton, Southampton, U.K., in 2018. He is currently pursuing the Ph.D. degree in computer science with the University of St Andrews, St Andrews, U.K.

His major interests are in the areas of federated learning, distributed machine learning, edge computing, model compression, and Internet of Things.



Rehmat Ullah received the Ph.D. degree in electronics and computer engineering from Hongik University, Seoul, South Korea, in 2020.

He is currently a Research Fellow with the University of St Andrews, St Andrews, U.K. Previously, he worked as a Research Fellow with Queen's University Belfast, Belfast, U.K., and as an Assistant Professor with Gachon University, Seongnam, South Korea. His research interests are in edge computing, information-centric networking, and 5G evolution and beyond with a recent focus on federated learning for edge computing systems. More information is available from www.rehmatkhan.com.

Paul Harvey received the Ph.D. degree in computing science from the University of Glasgow, Glasgow, U.K., in 2015.

He is one of the original founders of the Autonomous Networks Research and Innovation Lab, Rakuten Mobile, Tokyo, Japan, and the Co-Chair of the ITU Focus Group on autonomous networks. He is a Research Lead with Rakuten Mobile.

Peter Kilpatrick received the B.Sc. degree in mathematics and computer science in 1981 and the Ph.D. degree in computer science in 1985.

He is currently a Reader of Computer Science with Queen's University Belfast, Belfast, U.K. His interests include parallel programming models and cloud and edge computing.

Ivor Spence received the Ph.D. degree in computer science from Queen's University Belfast, Belfast, U.K., in 1984.

He did research on code generation with Queen's University Belfast, where he is currently a Reader of Computer Science and leads the artificial intelligence research theme. His research is primarily on heterogeneous computing systems for AI.

Blesson Varghese received the Ph.D. degree in computer science from the University of Reading, Reading, U.K., in 2011, on international scholarships.

He is a Reader of Computer Science with the University of St Andrews, St Andrews, U.K., and the Principal Investigator of the Edge Computing Hub. He is an Honorary Faculty Member of Queen's University Belfast, Belfast, U.K., and a previous Royal Society Short Industry Fellow. His interests include distributed systems that span the cloud-edge-device continuum and edge intelligence applications. More information is available from www.blessonv.com.