ARES: Adaptive Resource-Aware Split Learning for Internet of Things

Eric Samikwa[a,*], Antonio Di Maio[a] and Torsten Braun[a]

[a]Institute of Computer Science, University of Bern, Switzerland

ARTICLE INFO

Keywords:
Split learning
Internet of things
Distributed machine learning
Federated learning
Edge computing

ABSTRACT

Distributed training of Machine Learning models in edge Internet of Things (IoT) environments is challenging because of three main points. First, resource-constrained devices have large training times and limited energy budget. Second, resource heterogeneity of IoT devices slows down the training of the global model due to the presence of slower devices (*stragglers*). Finally, varying operational conditions, such as network bandwidth, and computing resources, significantly affect training time and energy consumption. Recent studies have proposed Split Learning (SL) for distributed model training with limited resources but its efficient implementation on the resource-constrained and decentralized heterogeneous IoT devices remains minimally explored. We propose Adaptive REsource-aware Split-learning (ARES), a scheme for efficient model training in IoT systems. ARES accelerates local training in resource-constrained devices and minimizes the effect of stragglers on the training through device-targeted split points while accounting for time-varying network throughput and computing resources. ARES takes into account application constraints to mitigate training optimization tradeoffs in terms of energy consumption and training time. We evaluate ARES prototype on a real testbed comprising heterogeneous IoT devices running a widely-adopted deep neural network and dataset. Results show that ARES accelerates model training on IoT devices by up to 48% and minimizes the energy consumption by up to 61.4% compared to Federated Learning (FL) and classic SL, without sacrificing the model convergence and accuracy.

1. Introduction

In conventional Machine Learning (ML) algorithms, models are usually deployed in a centralized setting, i.e. the data generated by end devices is collected and aggregated into a single server for training [1]. However, the raw data generated by Internet of Things (IoT) devices is very often private or sensitive and can be too large to transmit over the networks, which makes it infeasible to train a centralized model [2]. Distributed ML techniques are essential to allow training of joint/global models in a decentralized fashion, without requiring direct access to the raw data stored on the end devices [3]. Such techniques are of great appeal to benefit from the rich data yielded by distributed IoT devices to produce high quality models [4].

Federated Learning (FL) [5] is a distributed ML technique that allows end devices to participate in training a joint model on massively distributed data. Using only the data available to the client, the model is trained on the device and only the model update is sent over the network to the server for aggregation. Current state-of-the-art models such as Deep Neural Networks (DNN) contain millions of parameters, and have significant demands on memory, computation, and energy consumption [6, 7]. In FL, the computationally intensive training of the entire DNN model is executed on the device using the local data [5]. However, training the entire model on resource-constrained devices (e.g., IoT devices) leads to very large training times and device energy consumption [8]. Thus, the expensive computational demands of state-of-the-art models remain a serious impediment of ML deployment on distributed resource-constrained IoT devices.

Split Learning (SL) is a new ML technique that decouples model training from the need for direct access to the raw data, in which a model is split into at least two sub-models [9]. The client-side sub-model is trained on the client device where the local training data exists, and the server-side sub-model is trained on the server. In this process, the server does not have access to raw data from the client side [10]. An essential benefit of SL is that it greatly reduces the computation requirements on the client-side as it only computes a sub-model instead of the entire computationally demanding model. Thus, SL enables model training on devices with low computing resources such as IoT devices [11]. However, the efficient implementation of SL on resource-constrained and decentralised heterogeneous IoT devices has been minimally explored.

One key challenge of SL is how to determine the optimal distribution of the model training task among IoT devices and the server, in view of the varying network channel conditions, device and server computing resources, and the heterogeneous distributed IoT devices [12]. The decentralised training can severely benefit from techniques of reducing the amount of data transmitted over the networks and the computation burden on the resource-constrained devices, both in terms of training acceleration and energy consumption [13]. Such techniques are highly valuable when they have less significant impact on the resulting model accuracy. Specifically, this paper aims to address the following challenges for the efficient implementation of SL in IoT:

(1) *Stragglers arising from resource heterogeneity of IoT devices that slow down other devices during training:* IoT devices connected to a server for distributed model training may have heterogeneous computing and network resources.

*Corresponding author
eric.samikwa@inf.unibe.ch (E. Samikwa); antonio.dimaio@inf.unibe.ch (A.D. Maio); torsten.braun@inf.unibe.ch (T. Braun)

In a synchronous training process, *stragglers* are devices that require a longer time to complete a training round and slow down all other devices [14, 2]. This is because the model sharing process will need to wait until all devices have completed a training round [15]. Various studies have aimed at reducing the straggler problem using asynchronous training [15, 16]. However, these approaches often have an effect on the accuracy of the model as all devices may not contribute equally to training. Therefore, solutions that can reduce the effect of stragglers in distributed training with heterogeneous devices, typical in IoT systems, are essential.

(2) *Variable network throughput and computing resources on devices and server that affect the training time and energy consumption:* Due to time varying channel conditions, the network throughput between IoT devices and server can vary during training [17]. The available computing resources for executing server sub-models is impacted by varying server loads. The local sub-model execution time on devices is also impacted by variable background processes for the device. These factors have significant impact on the training time and energy consumption on the IoT devices [18, 19]. Therefore, adaptive resource-aware mechanisms that account for changing operational conditions are required for efficient practical SL.

(3) *Training time and energy consumption optimization tradeoffs:* Splitting the model is intuitively an ideal way of reducing the computation burden on IoT devices during training without a major impact on the resulting accuracy [20]. Although addressing issues 1) and 2) will provide ideal distribution of a model training task for a given set of hardware resources, they may not be optimal when application objectives or constraints are taken into account. For instance, the split point for accelerating the overall model training time may not always coincide with the split point to minimize energy consumption on IoT devices. Therefore, a dynamic mechanism that supports application requirements in cases where the tradeoffs (i.e. training time vs energy consumption) arise is beneficial.

In this paper, we present Adaptive REsource-aware Split-learning (ARES), a technique to mitigate the challenges (stragglers, variable network and computing resources, training optimization tradeoffs) for efficient model training in IoT. ARES accounts for the resource-constrained nature and heterogeneity of IoT devices. By generating optimized split vectors of device-targeted split points, ARES minimizes the negative impact of stragglers on the training process. ARES monitors the variable conditions during the training such as network throughput and computing resources on IoT devices and the server, and accordingly adapts the system split vector to accelerate training and minimize the total energy consumption. ARES takes into account the application time and energy constraints to enable efficient mitigation of optimization tradeoffs. Furthermore, ARES optimizes the training time and energy consumption on the devices without sacrificing the training convergence and accuracy.

In summary, this paper makes the following contributions.

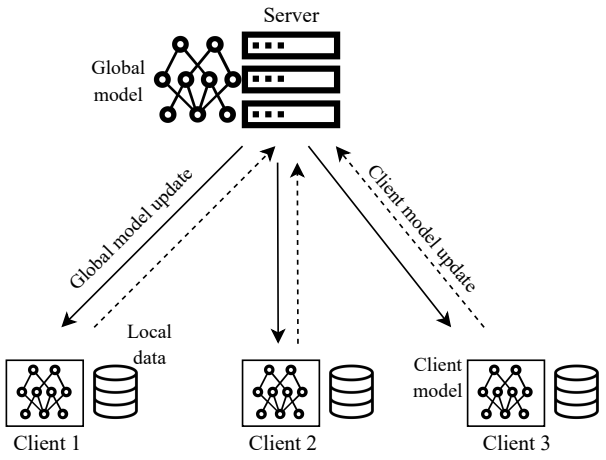

Figure 1: Federated learning representation with three clients.

- We propose ARES, a scheme for efficient decentralised model training in IoT systems. ARES jointly accelerates model training time and minimizes energy consumption in resource-constrained IoT devices through device-targeted split points while accounting for time-varying network throughput and computing resources on the devices and server. ARES minimizes the effects of stragglers on the training.
- We consider the tradeoffs between minimizing the training time and minimizing the energy consumption on IoT devices by introducing *energy-sensitivity* parameter for the training optimisation. The energy-sensitivity parameter is set by the application policy-maker and ensures the training optimization conforms to the application's energy or time constraints in cases of tradeoffs between the two objectives.
- We implement and evaluate ARES prototype on a real testbed comprising of heterogeneous IoT devices and an edge server. The evaluation results demonstrate the effectiveness of the proposed ARES scheme. ARES is available on public Github repository[1].

The rest of this article is organized as follows. Section 2 presents the relevant background and related work. Section 3 describes the assumptions about the scenario in which our solution operates and details the operation of our proposed solution. Section 4 provides details of the experimental testbed setup used to evaluate the proposed solution, the evaluation process, and highlights the results of the evaluation. Section 5 concludes the paper.

2. Background and Related Work

In the FL algorithm [5], model aggregation, Federated Averaging (FedAvg), is carried out on the server where a global model is generated by averaging local model updates (Figure 1). Over the entire training of FL, only the

[1] https://github.com/ricsamikwa/RES-Things

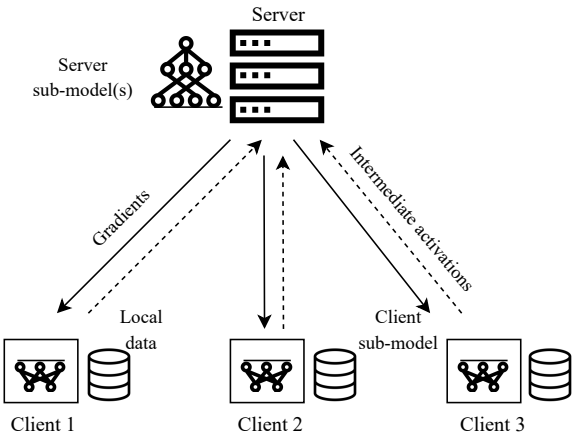

Figure 2: Split learning representation with three clients, where the model is split into two sub-models, namely client sub-model and server sub-model.

model parameters are communicated between clients and the server. Therefore, the clients do not require to share their raw training data to the server or to other clients. The local data remains local and confidential, which makes FL a privacy-preserving ML technique. However, training an entire model is often too computationally demanding and can lead to prolonged training times on resource-constrained devices [21]. Furthermore, FL is known to be less efficient for heterogeneous devices that have different computational capabilities, common in IoT systems. Thus, computation of the model on resource-constrained devices is a major bottleneck for the FL training process [22, 10, 23].

The SL training technique was first presented in [9]. From one specific layer, called *split layer* or cut layer, the neural network is split into two sub-networks. On the device-side, the model is trained up to the split layer. Then the activation feature maps of the split layer (*intermediate activations*) are sent to the server. The server continues training until the last layer of the model. After the training loss is calculated and the gradient is updated, the respective *intermediate gradients* of the split layer are sent to the device so that the gradients are updated on the device. An iteration of the above process over the entire client dataset is often referred to as *training round*. The training rounds are repeatedly carried out until the model converges.

Figure 1 shows a representation of the SL process, in which the server does not have direct access to raw data on clients. The advantage of SL compared to FL is that each client trains only a portion of the whole neural network, made of just a few layers, which considerably reduces the computational burden on clients [24]. This is essential for the proliferation of artificial intelligence enabled IoT, where ML methods enhance both applications and management of IoT networks [25]. However, basic SL does not take into account the computational heterogeneity of devices in IoT systems and the variable network channel conditions that affect data transmission throughput and computing resources on devices and server which, in turn, impact the total training time and energy consumption.

DNNs are comprised of sequence of layers with varying computational requirements and output sizes [18]. Splitting the model distributes the computation load of the model training task on the multiple devices involved. Furthermore, the split layer determines the amount of data transmitted over the network during the training task. For efficient distributed execution, the split layer for optimal performance needs to be identified. Depending on how the model is split, the time required to complete an SL training task and the energy consumption by the system to perform it can change [26]. Since the set of possible model split points is finite, the existence of a model splitting that minimizes training time or energy consumption or both is guaranteed. Since the training time and energy required for computation and transmission changes for each device, training time and energy optimal model splitting may not always coincide.

The available computing and network resources in IoT systems can change over time and affect both the training time and the energy consumption [26]. In high-network-throughput scenarios, for example, the IoT devices may take less time to transmit data from an intermediate layer while in low-network-throughput scenarios the time may be considerably higher. On the other hand, the available computing resources on the IoT devices and server can change due to third party processes. Therefore, the optimal split points may not always be the same for the entire life time of an application. The optimal splitting should then be selected according to variable training conditions and application requirements.

In comparison, model training in FL is relatively fast due to the parallel independent model updates in clients when the device local training is not a bottleneck (i.e. on more powerful devices). It has been shown that SL is more communication efficient with increasing the number of clients while FL becomes more efficient with increasing the number of data samples especially when the number of clients or the model size is small [27]. Based on the experiments for IoT setting [10], evaluating classification performance of the vanilla FL [5] and SL on Electrocardiography (ECG) and speech signals, SL achieves better classification accuracy than that of FL when data is Independent and Identically Distributed (IID), but less efficient under extreme non-IID data distribution. Thus, SL is a more suitable approach to support training in low resource environments and synergy of SL and FL techniques enables robust distributed training.

FedMask [21] allows computation and communication efficient personalised federated learning on heterogeneous devices. However, FedMask does not account for the varying network and computation conditions that affect the training, and resource-constrained nature of IoT devices. Furthermore, the process of creating neural network masks can have an effect on the resulting training accuracy.

Thapa et al. [28] proposed Split Federated Learning to achieve both parallel training and acceleration of device

Table 1
Comparison of ARES to FL and SL.

Characteristic	FL	SL	ARES
Decentralised training	✓	✓	✓
Independent (parallel) model updates	✓	×	✓
Resource-constrained devices	×	✓	✓
Energy consumption considerations	×	×	✓
Variable resources (network, computing)	×	×	✓
Training time and energy tradeoffs	×	×	✓

training. FedSL [29] combines parallel training and acceleration of device training for sequentially partitioned data in health applications. However, both approaches do not pay attention to the computation heterogeneity of the target devices and disregard energy consumption.

Wu et. al. [30] proposed FedAdapt, a mechanism for adaptive offloading for devices in FL. However, FedAdapt does not take into account differences in energy consumption for on-device processing and data transmission. Furthermore, these approaches do not pay attention to the optimization tradeoffs in view of user-defined constraints/objectives that can affect the outcomes of their methods.

Krouka et. al. [31] proposed model compression and splitting for energy efficient collaborative inference over time-varying channels. However, their approach affects the model accuracy for the inference operations. Similarly, early exit of inference mechanisms and offloading model layers from a device to a server to improve inference efficiency have been proposed in various studies (these do not consider training efficiency) [32, 33, 34, 35, 36].

Mao et. al. [37] proposed an approach for energy-efficient communication and computation over wireless powered mobile-edge system. The solution minimizes the total energy consumption of a cooperative edge system by jointly considering time-slot assignment, computation-task allocation, and the optimization of central processing unit frequencies for computing. However, their approach does not consider the efficiency of distributed model training on heterogeneous resource-constrained IoT devices using SL.

In our previous work, we introduced *Adaptive Early Exit of Computation (EEoC) for Energy-Efficient and Low-Latency Machine Learning over IoT Networks* [26]. EEoC is a method for distributed computation of ML inference over IoT networks for latency and energy optimization. EEoC considers the time varying network conditions and available computing resources to determine the optimal distribution of DNN models between the IoT device and edge by selecting an ideal model split point. EEoC improves inference efficiency for ML tasks in IoT systems but does not consider the training of the models using distributed training techniques such as SL.

Unlike the existing work, ARES accounts for the resource-constrained nature and heterogeneity of devices, typical in IoT systems. By enabling device-targeted split points for the various devices involved in the training, ARES minimizes the effects of stragglers on the training process. ARES utilizes realistic estimations of the variable conditions during the training (network throughput, computing resources on IoT devices and server). ARES adapts a split vector of device-specific split points for the IoT devices to jointly accelerates model training time and minimizes energy consumption in resource-constrained IoT devices while accounting for time-varying network throughput and computing resources on the devices and server. ARES optimizes the training time and energy consumption on the devices without affecting the training convergence and the resulting accuracy. Through an energy-sensitivity parameter, ARES takes into account application constraints to enable efficient mitigation of training optimization tradeoffs (i.e. training time, energy consumption). Table 1 presents a comparison of FL, SL and ARES.

3. Adaptive Resource-Aware Split Learning for Model Training in IoT Systems

3.1. System Model

We consider a scenario in which there are Φ heterogeneous IoT devices that cooperate with an edge server to perform training of a shared DNN model for all Φ IoT devices using each device's local dataset. The DNN model's architecture is composed of N consecutive layers, each denoted by L_i, with $i \in \{1, \dots, N\}$. We assume that each IoT device ϕ involved in the training process hosts a local dataset of size W_ϕ that does not change during the training. In the considered scenario, the training is performed using *minibatches* [38].We assume that the dataset on each IoT device ϕ is partitioned in *minibatches* of equal size, containing ξ entries each. The size ξ of each minibatch is defined as a training hyper-parameter and remains constant during training.

The training operates as follows. First, the model weights are initialized according to a weight initialization strategy. Then, for each of the ξ entries of the first minibatch, the model performs forward propagation and applies a loss function (e.g., sum of squares or cross entropy [39]) to the ξ outputs to accumulate the errors. The accumulated errors are used as the starting step of the Error Back Propagation (EBP) algorithm, which propagates error gradients back from the last layer L_N to the first layer L_1 to update the DNN weights. When the EBP algorithm is finished, all model weights have been updated according to the information contained in the minibatch. Now, the procedure of alternating forward and backpropagation to train the DNN model is repeated for each following minibatch until the model has "seen" all minibatches, i.e., the whole dataset. We define *training round* (or *epoch*) as the process of training the DNN over all minibatches (i.e., the whole dataset) one time, and we indicate each of them with k.

We consider SL where weight sharing is carried out by the system through aggregation of all model weight at the server. The frequency of carrying out weight sharing can vary depending on the local model convergence rate

Table 2
Symbol table

Symbol	Description
Φ	Number of devices in SL training task
R	Total number of rounds in SL the task
W_ϕ	Dataset size on device ϕ
ξ	Minibatch size for the training
N	Total number of layers of the model
L_i	i-th layer of the model
V_s	Total volume of intermediate activations from layer L_s of the model for the current minibatch of size ξ
V'_s	Total volume of intermediate gradients from layer L_{s+1} during back propagation for the current minibatch of size ξ
k	Training round k
$\delta_i^{(k)}(\phi)$	Forward propagation time for layer L_i on device ϕ at round k
$\delta'^{(k)}_i(\phi)$	Backward propagation time for layer L_i on device ϕ at round k
$D_s^{(k)}(\phi)$	Total forward and backward propagation times for a minibatch ξ on device ϕ at round k
$\gamma_i^{(k)}$	Forward propagation time for layer L_i on the server at round k
$\gamma'^{(k)}_i$	Backward propagation time for layer L_i on the server at round k
$C_s^{(k)}$	Total forward and backward propagation times for a minibatch ξ on the server at round k
$B_\phi^{(k)}$	Network throughput (transmitting) at round k for device ϕ
$B'^{(k)}_\phi$	Network throughput (receiving) at round k for device ϕ
$s_\phi^{(k)}$	Split point for device ϕ at round k
s_k	Round k split vector of device-specific split points
$\Delta_s^{(k)}(\phi)$	Training time at round k for device ϕ
$\Delta_k(s_k)$	System-wide training time at round k
$P_{\mathsf{c}}(\phi)$	Device ϕ power consumption during computation
$P_{\mathsf{t}}(\phi)$	Device ϕ power consumption during data transmission
$P_{\mathsf{r}}(\phi)$	Device ϕ power consumption during receiving
$E_s^{(k)}(\phi)$	Total energy consumption for device ϕ for round k
α	Energy sensitivity coefficient

or the type of application. This ensures that the models for all devices have the same performance at the end of the training and enhances convergence (similar technique applied in [40, 11]). The system trains the model over a number of consecutive rounds until the generalization error of the DNN is minimized.

In order to distribute the learning task between IoT devices and the edge server, the model is *split* in two *sub-models* for each device participating in the training. During training round k on an IoT device ϕ, the first $s_\phi^{(k)}$ layers $\{L_i | 1 \leq i \leq s_\phi^{(k)}, i \in \mathbb{N}\}$ are executed on the IoT device ϕ, while the last $N - s_\phi^{(k)}$ layers $\{L_i | s_\phi^{(k)} + 1 \leq i \leq N, i \in \mathbb{N}\}$

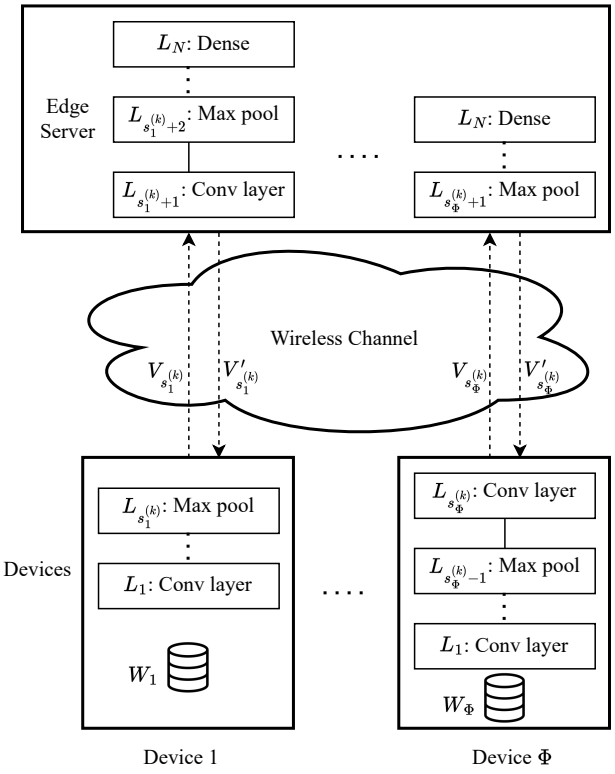

Figure 3: Split Learning between Φ IoT devices and an edge server where each IoT device ϕ has its own split point $s_\phi^{(k)}$.

are executed on the edge server. We call $s_\phi^{(k)}$ the model *split point* for the IoT device ϕ at training round k. We assume that during each training round k the IoT device ϕ can transfer data to the edge server with an average throughput $B_\phi^{(k)}$ and receive data from the edge server with an average throughput $B'^{(k)}_\phi$.

During forward propagation, an IoT device ϕ transmits a total volume $V_{s_\phi^{(k)}}$ of data generated by layer $L_{s_\phi^{(k)}}$ (*intermediate activations*) to the edge server for the minibatch of size ξ, so that it can continue the forward propagation from layer $L_{s_\phi^{(k)}+1}$. During backpropagation, the edge server transmits a total volume $V'_{s_\phi^{(k)}}$ of data generated by layer $L_{s_\phi^{(k)}+1}$ (*intermediate gradients*) to the IoT device ϕ for the minibatch of size ξ, so that it can continue updating the weights from layer $L_{s_\phi^{(k)}}$.

Figure 3 shows an example of Split Learning over Φ devices, where each device ϕ has its own split point $s_\phi^{(k)}$ and carries out model training with a local dataset of size W_ϕ. The sample model consists of five different DNN layers: convolution (Conv layer) that convolve the input to produce feature maps of inputs with the aim of learning features, max pooling (Max pool) apply a predefined function (maximum or average) to down sample the input, and dense layers (Dense) connect every neuron to all neurons in the previous layer with the aim of preforming high-leveled reasoning (usually stacked at the end of the model). Device Φ is

allocated more layers of the model than device 1, however, the number and sequence of layers is the same for all devices.

For every minibatch of size ξ, the intermediate activations produced by the layer $L_{s_\phi^{(k)}}$ at the split point are transferred via wireless communications with throughput $B_\phi^{(k)}$ to the layer $L_{s_\phi^{(k)}+1}$ on edge server. During backpropagation, the intermediate gradients produced by layer $L_{s_\phi^{(k)}+1}$ are sent back to layer $L_{s_\phi^{(k)}}$ on the IoT device through the same channel with throughput $B'^{(k)}_\phi$.

We consider that the computing resources on the edge server and on the IoT devices can change over time due to varying computation loads generated by third-party processes. We define $\delta_i^{(k)}(\phi)$ and $\delta'^{(k)}_i(\phi)$ as the time taken by the IoT device ϕ to execute forward and backward propagation, respectively, at layer L_i of the model during training round k for a single entry of the dataset. If the first s layers of the model run on the IoT device ϕ, the total time $D_s^{(k)}(\phi)$ to perform the training task for all entries of a minibatch of size ξ during training round k on the IoT device ϕ is the sum of the time required to perform forward and backward propagation on all layers between L_1 and L_s deployed on the IoT device ϕ, multiplied by the minibatch size ξ (Equation 1). The approach taken to identify the forward and backward propagation times $\delta_i^{(k)}(\phi)$ and $\delta'^{(k)}_i(\phi)$ on the devices is discussed in Section 3.3.

$$D_s^{(k)}(\phi) = \xi \sum_{i=1}^{s} \left(\delta_i^{(k)}(\phi) + \delta'^{(k)}_i(\phi) \right) \tag{1}$$

We now define $\gamma_i^{(k)}$ and $\gamma'^{(k)}_i$ as the time taken to execute forward and backward propagation, respectively, of layer L_i of the model on the edge server during training round k for a single entry of the dataset. If the last $N - s$ layers of the model run on the edge server, the total time $C_s^{(k)}$ to perform the training task for all entries of a minibatch of size ξ during training round k on the edge server is the sum of the time required to perform forward and backward propagation on all layers between L_{s+1} and L_N deployed on the edge server, multiplied by the minibatch size ξ (Equation 2).

$$C_s^{(k)} = \xi \sum_{i=s+1}^{N} \left(\gamma_i^{(k)} + \gamma'^{(k)}_i \right) \tag{2}$$

We model the transmission time of the intermediate activations of volume V_s for the current minibatch from the IoT device ϕ to the edge server over a wireless channel with bandwidth $B_\phi^{(k)}$, during training round k as $V_s / B_\phi^{(k)}$. We model the transmission time of the intermediate gradients of volume V'_s for the current minibatch from the the edge server to the IoT device ϕ over a wireless channel with bandwidth $B'^{(k)}_\phi$, during training round k as $V'_s / B'^{(k)}_\phi$. Once the edge server has completed forward propagation for all data entries in the minibatch it computes a loss function over all outputs, which takes a time $\Theta^{(k)}$ for each minibatch in training round k.

The process of forward propagation, intermediate data transmission, and backward propagation is sequential. Therefore, we can define the *training time* $\Delta_s^{(k)}(\phi)$ to complete the k-th training round on the IoT device ϕ when the model is split at layer L_s, as the sum of the minibatch-wise quantities defined so far (training times on the IoT device, training time on the edge server, transmission time from IoT device to edge server and back, and loss function computation on the edge server) divided by the minibatch size ξ (to obtain the average training time for each data entry), multiplied by the number W_ϕ of data entries in the dataset on ϕ (Equation 3). The variables upon which $\Delta_s^{(k)}(\phi)$ depends are either constant or variable depending on the computing and networking resources.

$$\Delta_s^{(k)}(\phi) = \frac{W_\phi}{\xi} \left(D_s^{(k)}(\phi) + C_s^{(k)} + \frac{V_s}{B_\phi^{(k)}} + \frac{V'_s}{B'^{(k)}_\phi} + \Theta^{(k)} \right) \tag{3}$$

In our considered scenario, the energy consumed by IoT device during the training process directly correlates with the device's lifetime before the battery must be replaced. This is an inconvenient operation that should be avoided. Therefore, it is important to model the energy consumption of IoT devices so that it will be possible to minimize it. On the other hand, we assume that the edge server does not have any restriction regarding energy consumption because the energy to run it is provided by the power grid and is assumed to be stable and inexpensive. In other words, we assume that the system-wide energy required to complete the SL training of a model between IoT devices and the edge server coincides with just the energy consumed by the IoT devices.

We define $P_{\mathrm{c}}(\phi)$ as the computing power consumption of the IoT device ϕ, $P_{\mathrm{t}}(\phi)$ as the power required by the IoT device ϕ to transmit data to the edge server over the wireless channel, and $P_{\mathrm{r}}(\phi)$ as the power required by the IoT device ϕ to receive data from the edge server. We assume that $P_{\mathrm{c}}(\phi)$, $P_{\mathrm{t}}(\phi)$, and $P_{\mathrm{r}}(\phi)$ are device-specific characteristics that do not change over time and that are known, for example through a benchmark to be performed offline.

It is worth noting that the energy required by device ϕ to perform forward and back propagation for a minibatch at training round k is the product between its computing power consumption $P_{\mathrm{c}}(\phi)$ and the time $D_s^{(k)}(\phi)$ needed to compute the activations and updated weights for the first s layers $\{L_i | 1 \leq i \leq s, i \in \mathbb{N}\}$ of the model for all entries of a minibatch of size ξ. We model the energy consumed by IoT device ϕ to transmit the intermediate activations generated by layer L_s for all entries of a minibatch of size ξ (with a total volume V_s) to the edge server through a wireless channel with upload throughput of $B_\phi^{(k)}$ bit/s as the product between the wireless upload power consumption $P_{\mathrm{t}}(\phi)$ and the time

needed to upload a volume V_s of data to the edge server ($V_s / B_\phi^{(k)}$). Similarly, we model the energy consumed by IoT device ϕ to receive the intermediate gradients generated by layer L_{s+1} for all entries of a minibatch of size ξ (with a total volume V'_s) from the edge server through a wireless channel with download throughput of $B'^{(k)}_\phi$ bit/s as the product between the wireless download power consumption $P_r(\phi)$ and the time needed to download a volume V'_s of data to the edge server ($V'_s / B'^{(k)}_\phi$).

We can now define the *total energy consumption* $E_s^{(k)}(\phi) \in \mathbb{R}$ as the energy consumed by the IoT device ϕ during the whole k-th training round of a model split at layer L_s. The quantity $E_s^{(k)}(\phi)$ is the sum of the energy consumed by the IoT device ϕ during forward and back propagation for all entries of a minibatch of size ξ during training round k for the first s layers, plus the energy needed to exchange the intermediate activations and gradients for all entries of the minibatch with the edge server, all multiplied by a normalization factor W_ϕ / ξ (Equation 4). The parameters $D_s^{(k)}(\phi)$, $B_\phi^{(k)}$, $B_\phi^{\prime(k)}$ can vary according to the available computing and networking resources.

$$E_s^{(k)}(\phi) = \frac{W_\phi}{\xi} \left(P_c(\phi) D_s^{(k)}(\phi) + P_t(\phi) \frac{V_s}{B_\phi^{(k)}} + P_r(\phi) \frac{V'_s}{B_\phi^{\prime(k)}} \right) \tag{4}$$

In our scenario, we assume that each of the Φ IoT devices has its own model split point $s_\phi^{(k)} \in \{1, \dots, N\}$, which can be different at each training round k according to the variable system context (e.g., available wireless throughput, computational load on IoT devices and edge server, etc.). We define the system *split vector* s_k as the vector of device-specific split points $s_k = \left(s_1^{(k)}, \dots, s_\phi^{(k)}\right)^\intercal \in S = \{1, \dots, N\}^\Phi$ for each IoT device ϕ during training round k. The symbol S identifies the space of all possible split vectors. At the end of each training round, the weights of the model's layers must be aggregated to obtain a global model for all IoT devices Φ (weight sharing). The edge server collects all sub models (i.e., their weights) from each IoT device ϕ via wireless channel and executes an aggregating function on them to obtain a single global model. This means that the edge server must wait until all Φ IoT devices have completed their k-th training round and have transmitted their sub models from before starting the aggregation procedure. After the sub-models from the IoT devices have been aggregated, they are distributed back to the IoT devices so that the training process can continue independently. Transferring the sub model parameters between the IoT device and the edge server also requires time and energy. However, for the scope of the present work, we assume that the size of the considered models corresponds to a negligible amount of time and energy expenditure in comparison to the actual training and, therefore, we do not consider it in the system model. We can now define the *system-wide training time* $\Delta_k(s_k)$ for training round k and split vector s_k as the maximum training time for all Φ IoT devices in the system (Equation 5). It is worth noting that reducing the training time of the slowest device (*straggler*) reduces the total training time for training round k.

$$\Delta_k(s_k) = \max_{\phi \in \{1, \dots, \Phi\}} \Delta^{(k)}_{s_\phi^{(k)}}(\phi) \tag{5}$$

Similarly, we define the *system-wide energy consumption* $E^{(k)}$ as the sum of the energy consumed by all IoT devices in the system during training round k (Equation 6).

$$E_k(s_k) = \sum_{\phi \in \Phi} E^{(k)}_{s_\phi^{(k)}}(\phi) \tag{6}$$

3.2. Problem Formulation

The main goal of ARES is to compute, for each training round k, a split vector s_k that jointly optimizes two objectives: minimizing the global model's training time and minimizing the energy consumption on the devices. Minimizing the global model training time Δ_k and minimizing the IoT devices' total energy consumption E_k are potentially conflicting objectives: for an IoT device ϕ, a split point that minimizes energy does not necessarily coincide with the split point that minimize global model training time [26]. Therefore, the actual value of the split vector that jointly optimizes model training time and energy depends on the considered application requirements. In particular, some applications may be energy-sensitive, meaning that extending the devices' lifetime is more important than reducing the global model training time. Some other applications may be more time-sensitive, meaning that minimizing training time is more important than extending device lifetime. To this end, we define the *energy sensitivity* $\alpha \in [0, 1]$ as a coefficient that represents the application's preferences between reducing training time or energy. We define α such that the closer α is to zero ($\alpha \to 0$) the lower the system's energy consumption. Conversely, the closer α is to one ($\alpha \to 1$) the shorter the global model's training time. In this work, we assume that the particular value of α does not change over time and is initially fixed by a policy maker that interprets the application's requirements.

This definition of α allows us to convert a multi-objective optimization problem, that aims at minimizing energy and training time separately, in a single-objective optimization problem. In particular, we design a *round cost function* $U_k(s_k) \in \mathbb{R}$, which depends on the split vector s_k at training round k, as a linear combination of the training time and energy consumption, mediated by the coefficient α (Equation 7).

$$U_k(s_k) = \alpha \cdot \Delta_k(s_k) + (1 - \alpha) \cdot E_k(s_k) \tag{7}$$

It is worth noting that when $\alpha = 1$ the round cost function will consider only the global model training time, while when $\alpha = 0$ the round cost function will consider only the system energy consumption, coherently with the meaning

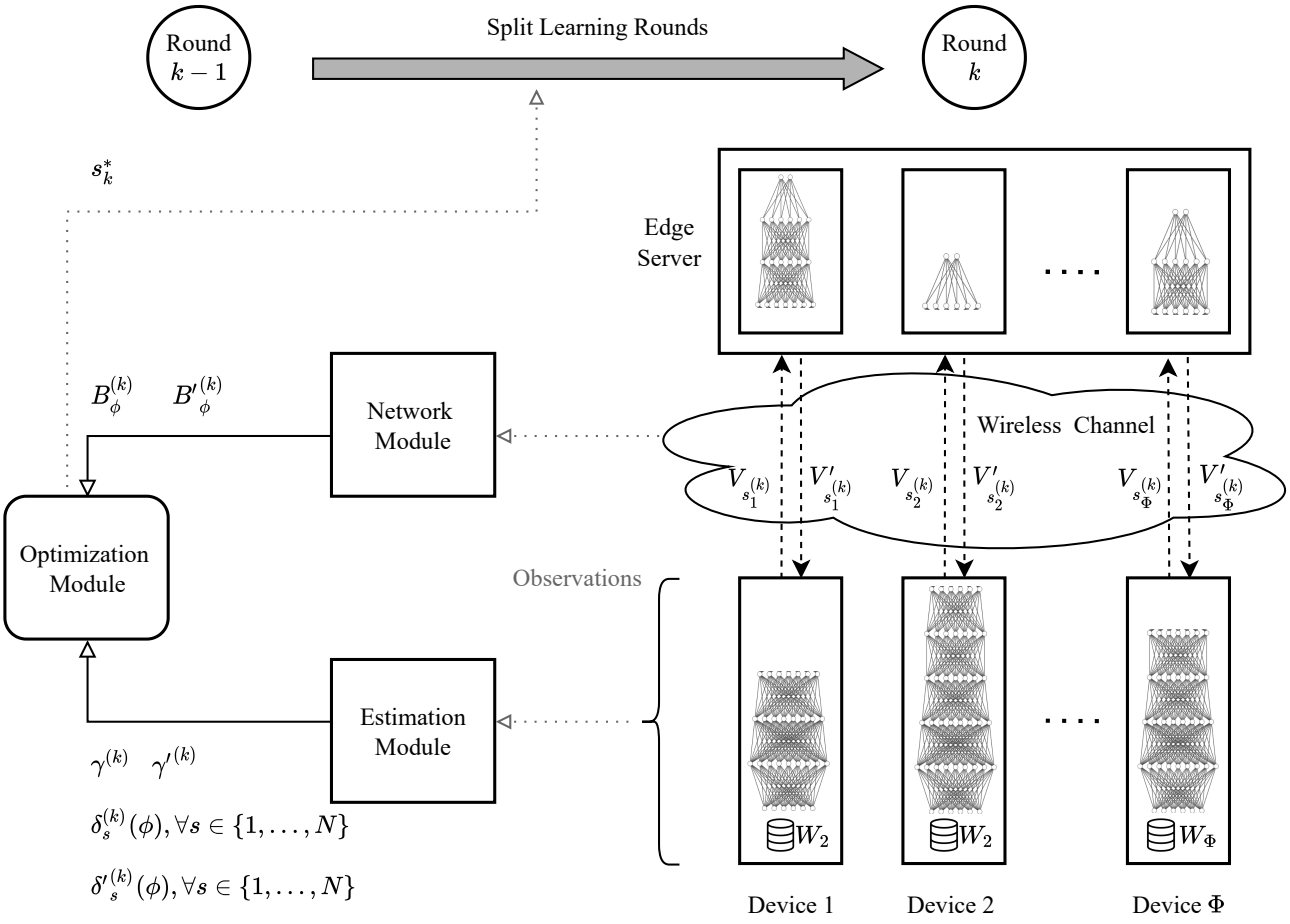

Figure 4: Adaptive REsource-aware Split-learning architecture, the network throughput is monitored at each training round k and computing resources information is periodically observed and utilized for training acceleration and minimizing energy consumption.

of the energy sensitivity coefficient α. For a training round k, the optimal split vector $s_k^* = \arg\min_{s_k \in S} \quad U_k(s_k)$ that minimizes the round cost function U_k achieves the best tradeoff between global model training time and system energy consumption. Let us now assume that the global model needs R training rounds for its generalization error to be minimized (*model convergence*). The ultimate goal of our proposed approach is to compute a split vector s_k for each of the R training rounds so that the global model training time and the system energy consumption until convergence is minimized. We introduce the *split matrix* $\sigma \in S^R = \{1,\dots,N\}^{\Phi \times R}$ as a matrix with Φ rows and R columns, in which the k-th column contains the elements of the split vector s_k. In other words, the element $\sigma_{\phi k}$ of the split matrix is the split point $s_\phi^{(k)}$ of IoT device ϕ at training round k. We also define a *general cost function* $U(\sigma) = \sum_{k=1}^{R} U_k(s_k)$ as the sum of the R round cost functions for every training round k until convergence. The *optimal split matrix* σ^* that minimizes the general cost function U, as presented in Equation 8, achieves the best tradeoff between global model training time and system energy consumption for all rounds until convergence.

$$\sigma^* = \arg\min_{\sigma \in S^R} \quad U(\sigma) \tag{8}$$

3.3. ARES Architecture and Operation

Our proposed scheme, ARES, is designed to compute the optimal split matrix σ^* over time, taking into account the variability of the system context. ARES operates through the interaction of three modules: the Estimation Module, the Network Module, and the Optimization Module. During each training round k, the Estimation Module and Network Module compute the information about the system context and then the Optimization Module computes an optimal split vector s_k that minimizes training time and energy consumption.

The Estimation Module is designed to periodically estimate the time needed by each layer $\{L_i | 1 \le i \le N, i \in \mathbb{N}\}$ of the model to perform forward and back propagation on the edge server and on each of the Φ IoT devices. At each training round k, an Estimation Module deployed on the IoT device ϕ estimates the quantities $\delta_s^{(k)}(\phi)$ and $\delta'^{(k)}_s(\phi), \forall s \in \{1,\dots,N\}$, which denote the time needed by the IoT device ϕ to execute forward and back propagation on each single layer $\{L_i | 1 \le i \le N, i \in \mathbb{N}\}$, respectively. Similarly, at

each training round k, an Estimation Module deployed on the edge server estimates the quantities $\gamma_s^{(k)}$ and $\gamma'^{(k)}_s, \forall i \in \{1, \dots, N\}$, which denote the time needed by the edge server to execute forward and back propagation on each single layer $\{L_i | 1 \leq i \leq N, i \in \mathbb{N}\}$, respectively. Both edge server and IoT devices estimate forward and back propagation time of each layer of the model through *benchmarking*: each Estimation Module deploys a copy of the entire global model in its local memory and performs multiple forward and back propagations on the local model while measuring the time needed by each layer to execute them. For each layer of the model, the mean time needed to perform forward and backward propagation is estimated by averaging the time measured for each of the benckmarking propagations and fed to the Optimization Module. In most cases, it is not necessary to update at every training round. Therefore, we assume that the Estimation Module performs a benchmark for IoT devices and edge server, respectively, every M_D and M_C training rounds and sets all the estimates for all training rounds between two consecutive benchmarks to the same values. Thus, M_D and M_C represent the device and server computation benchmark intervals respectively.

The Network Module is designed to periodically estimate the quantities $B_\phi^{(k)}$ and $B'^{(k)}_\phi, \forall \phi \in \{1, \dots, \Phi\}$, which denote the average throughput of the wireless channel from the IoT device ϕ to the edge server and vice versa, respectively, at each training round k. The Network Module is deployed and executed on the edge server. For each minibatch fed to model split at layer L_s during training round k, a volume V_s of intermediate activations is transmitted by the IoT device to the edge server and a volume V'_s of intermediate activations is transmitted by the edge server to the IoT device. For each minibatch, the Network Module measures the time needed to transfer the volume V_s of intermediate activations to the edge server and computes the average wireless channel throughput as the ratio of the volume V_s and the time needed to transfer it to the edge server. Similarly, for each minibatch, the Network Module measures the time needed to transfer the volume V'_s of intermediate gradients to the IoT device and computes the average wireless channel throughput as the ratio of the volume V'_s and the time needed to transfer it to the IoT device. At the end of the training round k, the Network Module estimated the average wireless channel throughput during the whole training round k as the average of the throughput values measured to transmit each minibatch's intermediate activations and gradients. Then, the uplink and downlink wireless channel throughput estimates are fed to the Optimization Module.

The Optimization Module takes as input the estimates from the Estimation Module and the Network Module and solves the optimization problem described in Equation 8 by exploring the split vector space S to find the optimal split vector s_k^* that globally minimizes the cost function U_k according to the current estimated system context for every training round k. The Optimization Module is deployed and executed on the edge server. After the optimal split vector s_k^* is computed, the edge server communicates the new split

Algorithm 1: ARES Operation

Input:
$\tau_i^{(k)}$: returns the average time needed to transmit activations from the i-th layer during round k
$f^{(k)}_\text{device}(L_i)$, $f^{(k)}_\text{edge}(L_i)$: return the average i-th layer's computation time on the IoT device and server
$\Omega : \mathbb{R} \to \mathbb{N}, \Omega(x) = \lfloor x/Q \rfloor$: discretizes throughput in equal intervals of Q bit/s
$\Gamma \in \mathbb{N}$: throughput measurement window size
M_D, M_C: device and server computation benchmark intervals
$\alpha \in [0, 1]$: energy sensitive coefficient
Output:
$s_k \in S = \{1, \dots, N\}^\Phi$: split vector for each round

`/* Initialization: throughput vectors */`
$b^{(1)} \leftarrow Q$
$B^{(0)} \leftarrow -Q$
`/* Loops for each round k */`
for $k \in R$ **do**
  if $k \mod M_D = 1$ **then**
    `/* Devices computation benchmark */`
    for $\phi \in \Phi$ **do**
      for $i = 1, \cdots, N$ **do**
        $\delta_i^{(k)}(\phi) \leftarrow f^{(k)}_\text{device}(L_i)$
        $\delta'^{(k)}_i(\phi) \leftarrow f'^{(k)}_\text{device}(L_i)$
  if $k \mod M_C = 1$ **then**
    `/* Edge server computation benchmark */`
    for $s = 1, \cdots, N$ **do**
      $\gamma_i^{(k)} \leftarrow f^{(k)}_\text{edge}(L_i)$
      $\gamma'^{(k)}_i \leftarrow f'^{(k)}_\text{edge}(L_i)$
  if $k > 1$ **then**
    $b^{(k)} \leftarrow V_{s^{(k-1)}} / \tau^{(k-1)}_{s^{(k-1)}}$
    $b'^{(k)} \leftarrow V'_{s^{(k-1)}} / \tau'^{(k-1)}_{s^{(k-1)}}$
  `/* Rolling window throughput estimations */`
  $j \leftarrow \max\{1, k - \Gamma + 1\}$
  $B^{(k)} \leftarrow \frac{1}{k-j+1} \sum_{i=j}^{k} b^{(i)}$
  $B'^{(k)} \leftarrow \frac{1}{k-j+1} \sum_{i=j}^{k} b'^{(i)}$
  `/* Changed resources? */`
  if $\Omega(B^{(k-1)}) = \Omega(B^{(k)})$ ***and*** $\Omega(B'^{(k-1)}) = \Omega(B'^{(k)})$ ***and*** $k \mod M_D \neq 1$ ***and*** $k \mod M_C \neq 1$ **then**
    $s^{(k)} \leftarrow s^{(k-1)}$
  else
    for $\phi \in \Phi$ **do**
      Compute $\{\Delta_1^{(k)}(\phi), \dots, \Delta_N^{(k)}(\phi)\}$
      Compute $\{E_1^{(k)}(\phi), \dots, E_N^{(k)}(\phi)\}$
    $s_k \leftarrow \arg\min \alpha \Delta^{(k)} + (1 - \alpha) E^{(k)}$

points to each of the Φ IoT devices in the system via the downlink wireless channel. Figure 4 shows ARES architecture, the interaction between its modules, the monitoring of the system context, and the updating of the split vector s_k at each training round k.

The overall operation of ARES and the order of interaction between its modules is described in Algorithm 1. At every training round k, ARES first checks whether the estimation module should perform benchmark. The estimation module determines the values $\delta_s^{(k)}(\phi)$ and $\delta'^{(k)}_s(\phi), \forall s \in \{1, \dots, N\}$, and $\gamma_s^{(k)}$ and $\gamma'^{(k)}_s, \forall i \in \{1, \dots, N\}$ for forward and back propagation times (from line 4 to 12). The periods for measuring the values, M_{D} and M_{C}, can be adjusted according to the variability of the available computing resources on the edge server or the IoT devices. Then, ARES estimates the average uplink and downlink throughput values $B_\phi^{(k)}$ and $B'^{(k)}_\phi, \forall \phi \in \{1, \dots, \Phi\}$ at every training round k between each IoT device ϕ and the edge server through rolling-window averages (lines 17 and 18). Using rolling-window averages increases the channel estimations accuracy because observing throughput changes in a window reduces the influence of noisy instantaneous variations of the throughput. For all Φ devices, if any benchmark has been performed during the current training round, or the throughput state has changed, the optimal split vector is recomputed, otherwise the old value is used (from lines 22 to 26).

4. Performance Evaluation

In this section, we evaluate the performance of ARES. First, we provide details, hardware specifications of the testbed setup and outline the tools used in implementing ARES prototype. Then, we present the procedure and results for the evaluation of ARES in terms of training time and energy consumption for various bandwidth and computing resources scenarios. Furthermore, we examine the model convergence and accuracy while applying ARES on the training process.

4.1. Experimental Testbed Setup

For all the performed experiments, we deployed a real testbed made of five heterogeneous IoT devices (2 Raspberry Pi 3B, 3 NVIDIA Jetson Nano) and one edge server, as shown in Figure 5. Table 3 outlines the specifications of testbed devices.

Each Jetson Nano is equipped with onboard power sensors, located at the power input of the board which can be read automatically with the `tegrastats` tool [2] or manually by reading Sysfs pseudo-file system [3] on Linux. We utilize the power measurements of Sysfs pseudo-file found in the I2C folder of the Jetson Nano. We utilize a power consumption benchmarking class with multithreading to monitor the power changes from the I2C during difference training operations on the Jetson Nano. We employ a plugin power monitor to measure the power consumption for Raspberry Pi.

All Raspberry Pis have the same version of Raspbian GNU/Linux 10 (Buster) operating system, Python version 3.7 and PyTorch version 1.4.0. All Jetson Nanos have the same version of NVIDIA JetPack SDK, software which

[2] https://github.com/topics/tegrastats
[3] https://man7.org/linux/man-pages/man5/sysfs.5.html

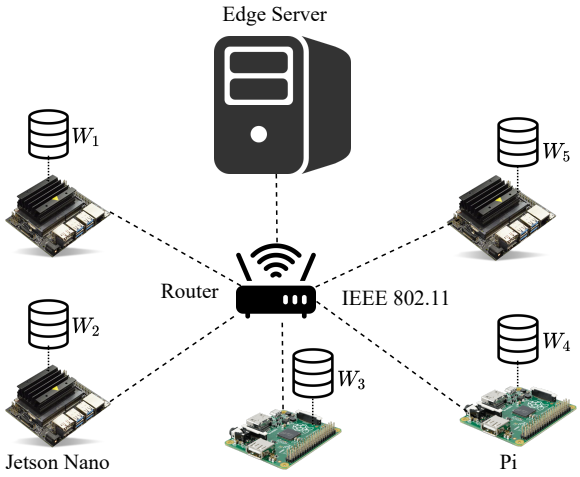

Figure 5: Overview of the testbed used for the evaluation.

Table 3
Testbed Specifications

Raspberry Pi 3B	
CPU	1.2 GHz core ARM Cortex-A53
RAM	900 MHz 1 GB LPDDR2
Operating System	Raspbian GNU/Linux 10 (Buster)
NVIDIA Jetson Nano	
CPU	Quad-core ARM Cortex-A57 MPCore
RAM	900 MHz 2 GB LPDDR4
Nano GPU	NVIDIA Maxwell arch. CUDA core
Operating System	NVIDIA JetPack SDK
	Jetson Linux Driver package (L4T)
Edge Server	
CPU	2.80 GHz 8-Core Intel i7-10510U
RAM	2666 MHz 16 GB DDR4
Operating System	Ubuntu 20.04.2 LTS

Table 4
Jetson Nano Power Modes

Modes	HIGH	MEDIUM	LOW
CPUS ONLINE	4	2	1
CPU MAX FREQ	1200 MHz	900 MHz	900 MHz
GPU MAX FREQ	900 MHz	600 MHz	600 MHz
MEM MAX FREQ	900 MHz	600 MHz	600 MHz

Table 5
Parameter table

Parameter	Value
Φ	5
N	10
R	100
ξ	100
W_ϕ	10,000
W	50,000
α	[0, 1]

includes the Jetson Linux Driver package (L4T), which provides the Linux kernel, NVIDIA drivers for manipulating hardware resources and power management. The Jetson Nanos and the server have the same version of Python and PyTorch. All devices are connected to the server in a network using a router. The IoT devices and the edge server can directly communicate via a IEEE 802.11 wireless link, whose available throughput can be controlled by a traffic shaping tool (Linux tc [4]).

We assume that all devices in the system cooperate to train a global model for image classification. We select the Visual Geometry Group (VGG) DNN model [41] as the global model deployed on the edge server and IoT devices. The VGG model is well-suited to assess a split learning scenario, as the compared algorithms can select a split point among 10 layers (VGG-8). We implement the considered model using Pytorch[5]. We selected VGG as the evaluation model because it is widely adopted in IoT applications based on edge intelligence. These include image classification in industrial IoT [42], smart security [4], smart vehicles [43], edge based speech commands processing [10], and smart health applications such as Electrocardiography [11], and COVID-19 screening [44]. Furthermore, SL models a DNN as a sequential connection of independently executable layers and not as an indivisible executable unit: therefore, our approach is generalizable and can be applied to any other sequential DNNs composed of any different arrangement of independently executable layers. This evaluation approach is commonly used in the related literature [31, 40, 33, 10].

We select the CIFAR-10 [45] dataset for training and testing. The CIFAR-10 dataset consists of 60k 32x32 colour images in 10 classes, with 6k images per class. The dataset contains a large set of 50k training and 10k testing labeled images. To generate a local dataset for each IoT device, we split the CIFAR-10 dataset in five disjoint partitions of size $W_1 = \ldots = W_5$ and then we load each of them on a different IoT devices. Table 5 shows a summary of the general parameters used in the evaluation.

[4]https://man7.org/linux/man-pages/man8/tc.8.html
[5]https://pytorch.org/

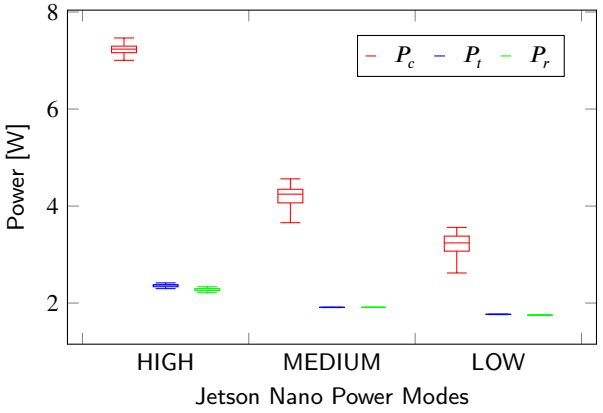

Figure 6: Computing, network transmission and receiving power for the three Jetson Nano configurations (power modes)

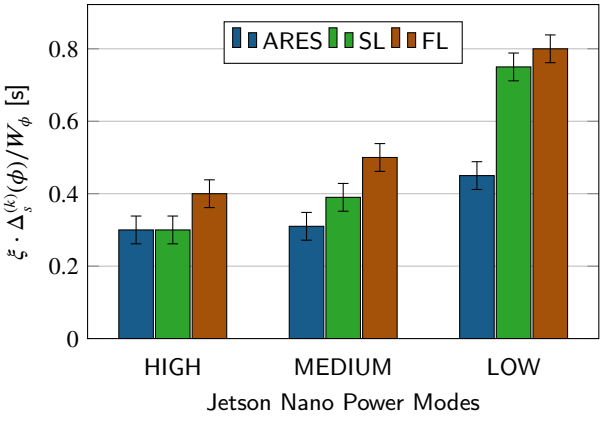

Figure 7: Average training time on Jetson Nano for minibatch with size $\xi = 100$.

4.2. Variable Computing Resources and Power Monitoring

We utilize the NVIDIA drivers for manipulating hardware resources and power management on Jetson Nano. Available computing resources on the Jetson Nano are modified by tuning the power modes. The JetPark software offers two default power modes and the possibility for custom power modes using the NVIDIA drivers. We define three configurations of CPU, GPU, and Memory clock speed to represent three classes of power consumption modes, namely HIGH, MEDIUM and LOW. The corresponding hardware resource configurations are shown in Table 4. We monitored the parameters P_c, P_t, and P_r through running separated computing (forward and back propagation) and recorded power readings from the I2C sensors for the different operations. Figure 6 shows the power consumption of the Jetson Nano for computing, transmitting and receiving for the three power mode configurations (HIGH, MEDIUM, LOW).

We examine the performance of ARES in a scenario with variable computing resources. To achieve this, we observe the performance of ARES under the three power modes of the Jetson Nano for their variability in the computing

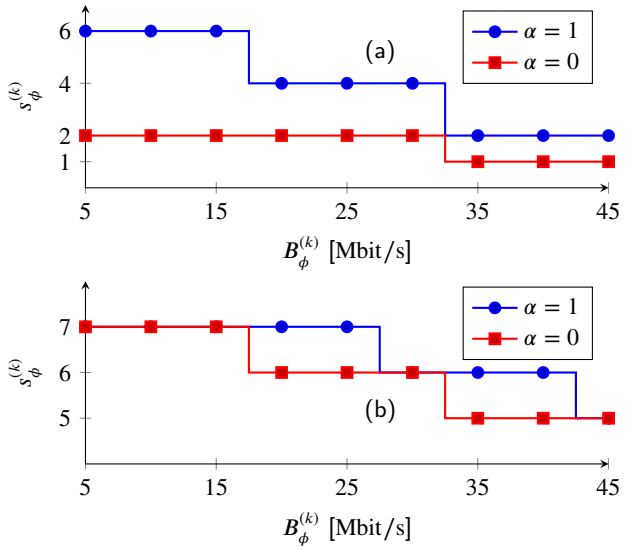

Figure 8: Optimal split point $s_\phi^{(k)}$ computed by ARES for the Raspberry Pi (a) and Jetson Nano (b) for different link throughputs $B_\phi^{(k)}$.

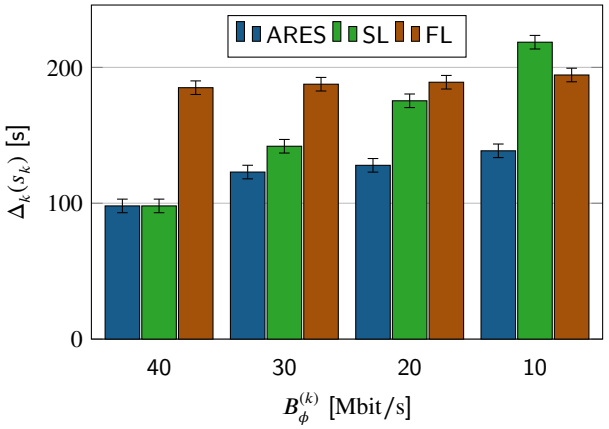

Figure 9: Average training time per round for the system with energy sensitivity coefficient set at $\alpha = 1$.

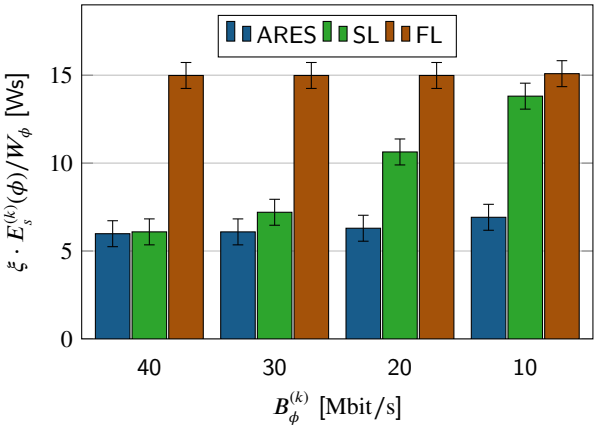

Figure 10: Average energy consumption for training minibatch with size $\xi = 100$ on Raspberry Pi with $\alpha = 0$.

resources. We monitor the average minibatch training time $\xi \cdot \Delta^{(k)}/W_\phi$ for a minibatch of size ξ on the Jetson Nano for the three scenarios with different available resources. We set the minibatch size ξ to be 100 for all the training evaluation scenarios. We compare the average minibatch training time achieved by ARES to the average average minibatch training time for FL and SL. Figure 7 shows the average training time for a minibatch of the ARES, FL, and SL approaches under difference computing resources on the Jetson Nano. The confidence intervals are set at 95% minimum significance level. ARES achieves the lowest average minibatch training time compared to FL and SL for all the three power modes. The best performance is observed for the LOW power mode configuration with 44% and 41% reduction in training time compared to FL and SL respectively.

4.3. Training Time and Energy Optimization for Heterogeneous Devices

We examine the optimal split points for minimizing training time and energy consumption for the two device types: Raspberry Pi and Jetson Nano. We adjust the available network throughput between the device and the server for the range 5 Mbit/s to 45 Mbit/s and observe the split point for minimizing training time and energy consumption by setting the energy sensitivity coefficient α to 0 and 1 for always optimal training time and energy consumption respectively.

Figure 8 shows the optimal split point $s_\phi^{(k)}$ computed by ARES for the Raspberry Pi (a) and Jetson Nano (b) for different link throughputs $B_\phi^{(k)}$ during training round k. The two curves correspond to two extreme and opposite policies, namely energy minimization ($\alpha = 0$) and training time minimization ($\alpha = 1$). The curves corresponding to any other value of α always are between the two extreme curves. For both cases, the optimal split points remain constant or change depending on the network throughput changes.

The optimal split point for minimizing training time $\alpha = 1$ and minimizing energy consumption $\alpha = 0$ are is the same for some cases while different in most cases. This happens because the link throughput impacts the transmission and receiving time and consequently the energy consumption and the training time. Furthermore, the differences in computation power P_c, transmission power P_t and receiving power P_r for the considered IoT device affects the variability of the split point when optimizing for training time and energy consumption. This signifies the importance of the energy sensitivity coefficient α to mitigate the tradeoff between optimizing for training time and energy consumption depending on the application requirements. Depending on the application requirements, the coefficient α can be modified to suit the scenario, for example in the event where the training on the device is carried out at night when the IoT system is not in use, the value of α can be set to optimize more towards energy consumption rather than time.

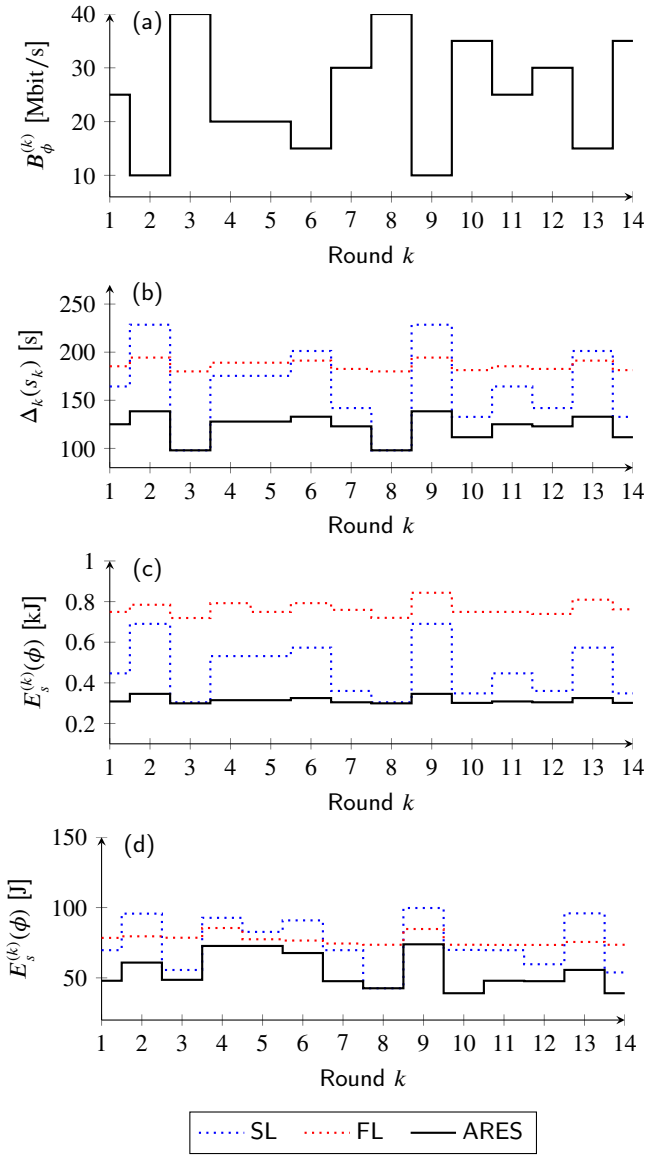

Figure 11: Impact of the system context on the (b) training time $\Delta^{(k)}$, (c) energy consumption $E_s^{(k)}(\phi)$ on the Raspberry Pi, and the (d) energy consumption $E_s^{(k)}(\phi)$ on the Jetson Nano to complete training the VGG model at each round k.

It is worth noting that the optimal split points on Raspberry Pi (a) and Jetson Nano (b) are not the same for corresponding network throughput values. The split points on the Jetson Nano are on average higher in the model than the split points on the Raspberry Pi. This means ARES allocates more training computation on the device for the relatively powerful Jetson Nano compared to the Raspberry Pi. This demonstrates the capacity of ARES to generate device-targeted split vector and to minimize the effect of stragglers on the training process.

4.4. Adaptability to Changing Network Conditions

We examine the training time and energy consumption optimization of ARES through different available network throughput. First, we adjust the available network throughput between the IoT devices and the server for the range 10 Mbit/s to 40 Mbit/s and observe the training time optimization by ARES with the energy sensitivity coefficient α set to 1. Figure 9 shows the results for the evaluation. ARES achieves the lowest average training time compared to FL and SL for the all throughput levels. ARES reduces the training time up to 48% and 37.5% compared to FL and SL respectively. Then, we observe the energy consumption on the Raspberry Pi to complete training a minibatch of size ξ through the network throughput changes (10 Mbit/s to 40 Mbit/s).

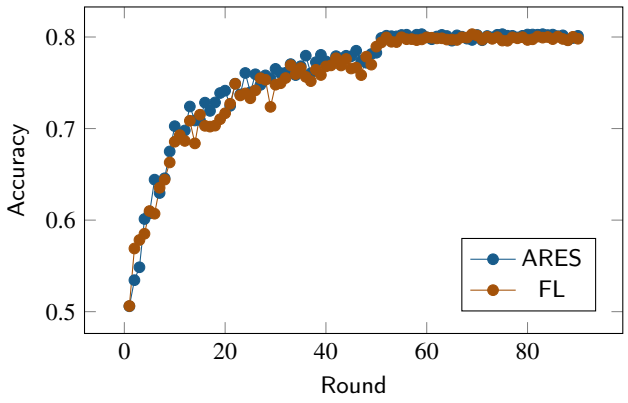

Figure 12: Model convergence and test accuracy for ARES and FL for VGG.

Figure 10 shows the average energy consumption $\xi \cdot \Delta^{(k)}/W_\phi$ by the Raspberry Pi to complete a minibatch training. ARES achieves the lowest average energy consumption compared to FL and SL for the all throughput levels. ARES reduces the average energy consumption up to 62.8% and 49.8% compared to FL and SL respectively.

We further investigate the selected solutions' *adaptiveness*, i.e., how their performance changes according to variations in link throughput over time. Therefore, we utilize the traffic shaper to set the network throughput, for every round k, to a random value extracted by a uniform distribution from 10 Mbit/s to 40 Mbit/s during the training. The ARES energy sensitivity coefficient is set at $\alpha = 0.5$.

Figure 11 shows the impact of the system context on the training time and energy consumption of IoT devices at each training round k. The (a) link throughput $B^{(k)}$ changes at each round k, following a uniform distribution from 10 Mbit/s to 40 Mbit/s. The (b) training time $\Delta^{(k)}$, (c) energy consumption $E_s^{(k)}(\phi)$ on the Raspberry Pi, and the (d) energy consumption $E_s^{(k)}(\phi)$ on the Jetson Nano to complete a training for the VGG model at each round k. The energy consumption by the Jetson Nano and Raspberry Pi are shown separately in accordance the heterogeneity. ARES achieves the lowest average energy consumption compared to FL and SL for the all throughput levels. ARES reduces minimizes training time and energy consumption up to 48% and 61.4% respectively compared to FL and SL.

4.5. Model Accuracy Comparison: ARES and FL

Model accuracy is a key aspect of every training operation, hence each training optimisation methods needs to take into consideration the convergence of the model and its accuracy. We monitored the resulting test accuracy and the convergence rate for the VGG model for ARES and FL approaches. Figure 12 compares the test accuracy of ARES and classic FL for 90 rounds. ARES achieves the same convergence speed and final accuracy as classic FL while optimizing the training time and energy consumption. Distributed model training techniques are highly beneficial when they have less significant impact on the resulting model accuracy since the training optimization does not compromise the integrity of the IoT system.

5. Conclusion

Implementing ML in edge IoT environments remains attractive for achieving robust and reliable intelligence in IoT systems. Studies have proposed SL for distributed model training with limited resources in IoT systems. However, the heterogeneity of IoT devices, variable network and computing resources, application constraints and training optimization tradeoffs are major challenges for efficient implementation of SL in IoT. We presented ARES, a scheme for efficient model training in IoT systems. ARES jointly accelerates model training time and minimizes energy consumption in resource-constrained IoT devices and minimizes the effects of stragglers on the training through device-targeted split points while accounting for time-varying network throughput and computing resources. ARES takes into account application constraints/objectives to mitigate training optimization tradeoffs in terms of minimizing energy consumption and training time. We evaluated ARES prototype on a real testbed comprising of heterogeneous resource-constrained IoT devices. ARES accelerates model training time on IoT devices up to 48% and minimizes the energy consumption up to 61.4% compared to FL and SL, without sacrificing the model convergence and accuracy.

Limitations and Future Work: our proposed scheme can be combined with existing model compression techniques such as pruning for more efficient distributed training in resource-constrained IoT devices. ARES can also be further extended to determine whether an alternate model can be selected to improve the training performance instead of partitioning only a given DNN, in scenarios with wide heterogeneity of network and computing resources. Exploring techniques of reducing the communication overhead during the training is another area of interest since SL relies on communication between IoT devices and server. Techniques such as parameter quantization and adaptive gradient threshold mechanisms may reduce the communication cost.

References

[1] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[2] Ahmed Imteaj, Urmish Thakker, Shiqiang Wang, Jian Li, and M Hadi Amini. Federated learning for resource-constrained iot devices: Panoramas and state-of-the-art. *arXiv preprint arXiv:2002.10610*, 2020.

[3] Aluizio F Rocha Neto, Flavia C Delicato, Thais V Batista, and Paulo F Pires. Distributed machine learning for iot applications in the fog. *Fog Computing: Theory and Practice*, pages 309–345, 2020.

[4] Farzad Samie, Lars Bauer, and Jörg Henkel. From cloud down to things: An overview of machine learning in internet of things. *IEEE Internet of Things Journal*, 6(3):4921–4934, 2019.

[5] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[6] Zhongnan Qu, Cong Liu, Junfeng Guo, and Lothar Thiele. Deep partial updating. *arXiv preprint arXiv:2007.03071*, 2020.

[7] Zhongnan Qu, Zimu Zhou, Yun Cheng, and Lothar Thiele. Adaptive loss-aware quantization for multi-bit networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7988–7997, 2020.

[8] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[9] Otkrist Gupta and Ramesh Raskar. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116:1–8, 2018.

[10] Yansong Gao, Minki Kim, Sharif Abuadbba, Yeonjae Kim, Chandra Thapa, Kyuyeon Kim, Seyit A Camtepe, Hyoungshick Kim, and Surya Nepal. End-to-end evaluation of federated learning and split learning for internet of things. *arXiv preprint arXiv:2003.13376*, 2020.

[11] Yansong Gao, Minki Kim, Chandra Thapa, Sharif Abuadbba, Zhi Zhang, Seyit A Camtepe, Hyoungshick Kim, and Surya Nepal. Evaluation and optimization of distributed machine learning techniques for internet of things. *arXiv preprint arXiv:2103.02762*, 2021.

[12] Wiebke Toussaint and Aaron Yi Ding. Machine learning systems in the iot: Trustworthiness trade-offs for edge intelligence. In *2020 IEEE Second International Conference on Cognitive Machine Intelligence (CogMI)*, pages 177–184. IEEE, 2020.

[13] Stefano Savazzi, Monica Nicoli, and Vittorio Rampa. Federated learning with cooperating devices: A consensus approach for massive iot networks. *IEEE Internet of Things Journal*, 7(5):4641–4654, 2020.

[14] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. Asynchronous online federated learning for edge devices with non-iid data. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 15–24. IEEE, 2020.

[15] Zirui Xu, Fuxun Yu, Jinjun Xiong, and Xiang Chen. Helios: Heterogeneity-aware federated learning with dynamically balanced collaboration. *arXiv preprint arXiv:1912.01684*, 2019.

[16] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. Taming momentum in a distributed asynchronous environment. *arXiv preprint arXiv:1907.11612*, 2019.

[17] Prince Abudu and Andrew Markham. Learning distributed communication and computation in the iot. *Computer Communications*, 161:150–159, 2020.

[18] Luke Lockhart, Paul Harvey, Pierre Imai, Peter Willis, and Blesson Varghese. Scission: Context-aware and performance-driven edge-based distributed deep neural networks. *arXiv e-prints*, pages arXiv–2008, 2020.

[19] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, 2020.

[20] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, and Seyit A Camtepe. Advancements of federated learning towards privacy preservation: from federated learning to split learning. In *Federated Learning Systems*, pages 79–109. Springer, 2021.

[21] Ang Li, Jingwei Sun, Xiao Zeng, Mi Zhang, Hai Li, and Yiran Chen. Fedmask: Joint computation and communication-efficient personalized federated learning via heterogeneous masking. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 42–55, 2021.

[22] Cong Wang, Yuanyuan Yang, and Pengzhan Zhou. Towards efficient scheduling of federated mobile devices under computational and statistical heterogeneity. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):394–410, 2020.

[23] Qinbin Li, Zeyi Wen, Zhaomin Wu, Sixu Hu, Naibo Wang, Yuan Li, Xu Liu, and Bingsheng He. A survey on federated learning systems: vision, hype and reality for data privacy and protection. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[24] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *arXiv preprint arXiv:2103.04505*, 2021.

[25] Hao Song, Jianan Bai, Yang Yi, Jinsong Wu, and Lingjia Liu. Artificial intelligence enabled internet of things: Network architecture and spectrum access. *IEEE Computational Intelligence Magazine*, 15(1):44–51, 2020.

[26] Eric Samikwa, Antonio Di Maio, and Torsten Braun. Adaptive early exit of computation for energy-efficient and low-latency machine learning over iot networks. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, pages 200–206. IEEE, 2022.

[27] Abhishek Singh, Praneeth Vepakomma, Otkrist Gupta, and Ramesh Raskar. Detailed comparison of communication efficiency of split learning and federated learning. *arXiv preprint arXiv:1909.09145*, 2019.

[28] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, and Seyit Camtepe. Splitfed: When federated learning meets split learning. *arXiv preprint arXiv:2004.12088*, 2020.

[29] Ali Abedi and Shehroz S Khan. Fedsl: Federated split learning on distributed sequential data in recurrent neural networks. *arXiv preprint arXiv:2011.03180*, 2020.

[30] Di Wu, Rehmat Ullah, Paul Harvey, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. Fedadapt: Adaptive offloading for iot devices in federated learning. *arXiv preprint arXiv:2107.04271*, 2021.

[31] Mounssif Krouka, Anis Elgabli, Chaouki Ben Issaid, and Mehdi Bennis. Energy-efficient model compression and splitting for collaborative inference over time-varying channels. *arXiv preprint arXiv:2106.00995*, 2021.

[32] Enzo Baccarelli, Michele Scarpiniti, Alireza Momenzadeh, and Sima Sarv Ahrabi. Learning-in-the-fog (lifo): Deep learning meets fog computing for the minimum-energy distributed early-exit of inference in delay-critical iot realms. *IEEE Access*, 9:25716–25757, 2021.

[33] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2019.

[34] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. *arXiv preprint arXiv:2006.04152*, 2020.

[35] Liekang Zeng, En Li, Zhi Zhou, and Xu Chen. Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things. *IEEE Network*, 33(5):96–103, 2019.

[36] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[37] Sun Mao, Jinsong Wu, Lei Liu, Dapeng Lan, and Amir Taherkordi. Energy-efficient cooperative communication and computation for wireless powered mobile-edge computing. *IEEE Systems Journal*, 2020.

[38] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[39] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[40] Ahmad Ayad, Melvin Renner, and Anke Schmeink. Improving the communication and computation efficiency of split learning for iot applications. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 01–06. IEEE, 2021.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv e-prints arXiv:1409.1556*, 2014.

[42] Xing Liu, Wei Yu, Fan Liang, David Griffith, and Nada Golmie. Toward deep transfer learning in industrial internet of things. *IEEE Internet of Things Journal*, 8(15):12163–12175, 2021.

[43] Zhongqin Bi, Ling Yu, Honghao Gao, Ping Zhou, and Hongyang Yao. Improved vgg model-based efficient traffic sign recognition for safe driving in 5g scenarios. *International Journal of Machine Learning and Cybernetics*, 12(11):3069–3080, 2021.

[44] Harshit Kaushik, Dilbag Singh, Shailendra Tiwari, Manjit Kaur, Chang-Won Jeong, Yunyoung Nam, and Muhammad Attique Khan. Screening of covid-19 patients using deep learning and iot framework. *Cmc-Computers Materials & Continua*, pages 3459–3475, 2021.

[45] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Eric Samikwa is a Ph.D. candidate at the Communication and Distributed Systems (CDS) group, Institute of Computer Science, University of Bern, Switzerland. He received his MSc in computer science and engineering from the Royal Institute of Technology (KTH), Sweden, and BSc from University of Malawi. His research interests are in the areas of distributed machine learning, split learning, edge computing, and the Internet of things.

Antonio Di Maio is a postdoctoral researcher in mobile networks with the Communication and Distributed Systems (CDS) group at the University of Bern, Switzerland. He obtained his PhD degree in Computer Engineering from the University of Luxembourg in 2020, with a thesis on routing and content dissemination in software-defined vehicular networks. His current research interests fall within the areas of network modeling, scheduling, routing, and channel access.

Torsten Braun is currently director at the Institute of Computer Science, University of Bern, where he has been a full professor since 1998. He got the Ph.D. degree from University of Karlsruhe (Germany) in 1993. From 1994 to 1995, he was a guest scientist at INRIA Sophia-Antipolis (France). From 1995 to 1997, he worked at the IBM European Networking Centre Heidelberg (Germany) as a project leader and senior consultant. He has been a vice president of the SWITCH (Swiss Research and Education Network Provider) Foundation from 2011 to 2019. He has been a Director of the Institute of Computer Science and Applied Mathematics at University of Bern between 2007 and 2011, and from 2019 to 2021.