Review and Comparative Evaluation of Resource-Adaptive Collaborative Training for Heterogeneous Edge Devices

BORIS RADOVIČ, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia and University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia
MARCO CANINI, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
VELJKO PEJOVIĆ, University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia and Jozef Stefan Institute, Ljubljana, Slovenia

Growing concerns about centralized mining of personal data threatens to stifle further proliferation of machine learning (ML) applications. Consequently, a recent trend in ML training advocates for a paradigm shift – moving the computation of ML models from a centralized server to a federation of edge devices owned by the users whose data is to be mined. Though such decentralization aims to alleviate concerns related to raw data sharing, it introduces a set of challenges due to the hardware heterogeneity among the devices possessing the data. The heterogeneity may, in the most extreme cases, impede the participation of low-end devices in the training or even prevent the deployment of the ML model to such devices.

Recent research in distributed collaborative machine learning (DCML) promises to address the issue of ML model training over heterogeneous devices. However, the actual extent to which the issue is solved remains unclear, especially as an independent investigation of the proposed methods' performance in realistic settings is missing. In this paper, we present a detailed survey and an evaluation of algorithms that aim to enable collaborative model training across diverse devices. We explore approaches that harness three major strategies for DCML, namely Knowledge Distillation, Split Learning, and Partial Training, and we conduct a thorough experimental evaluation of these approaches on a real-world testbed of 14 heterogeneous devices. Our analysis compares algorithms based on the resulting model accuracy, memory consumption, CPU utilization, network activity, and other relevant metrics, and provides guidelines for practitioners as well as pointers for future research in DCML.

CCS Concepts: • **Computing methodologies → Machine learning**; **Distributed artificial intelligence**.

Additional Key Words and Phrases: Federated Learning, Split Learning, Distributed Collaborative Learning, Ubiquitous and Mobile Computing, Device Heterogeneity.

1 Introduction

The surge in Machine Learning (ML) applications we have witnessed in the last years has been rendered possible, among other factors, by the increased computational capacity of modern hardware and the large volumes of data, that have become publicly available. The former is highly concentrated in data centers, as the devices that often collect the data, such as smartphones and IoT devices, have orders of magnitude lower computational power and storage capacities. Consequently, the traditional workflow for ML has remained centralized in the sense that the data is fully revealed and accessible by machines performing model training and evaluation.

Authors' Contact Information: Boris Radovič, King Abdullah University of Science and Technology, Thuwal, Makkah, Saudi Arabia and University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia; e-mail: boris.radovic@kaust.edu.sa; Marco Canini, King Abdullah University of Science and Technology, Thuwal, Makkah, Saudi Arabia; e-mail: marco@kaust.edu.sa; Veljko Pejović, University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia and Jozef Stefan Institute, Ljubljana, Slovenia; e-mail: veljko.pejovic@fri.uni-lj.si.

ACM 2376-3647/2024/12-ART
https://doi.org/10.1145/3708983

In many cases, however, data privacy is of paramount importance. For instance, a wealth of personal data such as sensor readings, images, and text, is typically stored on users' smartphones. The usage of this data could open space for numerous innovative applications, ranging from text auto-completion to personalized health monitoring systems. However, despite the potential value of these applications, users are typically hesitant to expose their data due to privacy and data ownership concerns [58]. Users' unwillingness to share their private data is evidenced by directives introduced by many countries to govern how companies can collect and store user data [80]. To a certain extent, data anonymization approaches might allow us to circumvent these issues and hence permit the usage of centralized approaches even in privacy-sensitive use cases. Yet, they do so at the expense of a significant computational overhead [32, 88], increased network usage [16], and decreased model performance [88].

Given the central role of smartphones and other data-collecting devices in contemporary society, a more sensible approach than imposing some form of centralization of the ML workflow involves designing algorithms specifically crafted to ensure privacy-sensitive operations. Such a "decentralization" shift is further motivated by edge devices' frequent requirement to independently conduct inference on data, such as when operating offline.

In response to these concerns and challenges, the concept of Distributed Collaborative Machine Learning (DCML) has emerged. In this work, we define DCML as an umbrella term that encompasses all algorithms designed *a)* to train models using data distributed across a set of devices without exposing the training data to any server, and *b)* to distribute an appropriate model to the devices that request it. Notably, the advent of Federated Learning (FL) and its seminal algorithm, FedAvg [58], have been pivotal in this domain. In fact, FL has garnered significant attention and several companies utilize it to train models on users' smartphones while safeguarding privacy. Concrete examples include various features in Android, such as next-word prediction and smart reply [33, 35], and Apple's "Siri" voice assistant [100].

FL algorithms enable a privacy-preserving training process by moving the computation of model updates to the data-collecting devices (also called "clients"). That is, a centralized server only orchestrates the training process by *a)* selecting the clients to be used for training in the current server training round, *b)* serving to these clients the current model, and *c)* aggregating the models returned by the clients after they finish training. This way, the server obtains a refined model, that will be used in the next training round.

Within this general formulation, the FedAvg [58] algorithm and variants thereof [53, 72] require each client to download and upload a complete version of the model at every server round. However, while such model-sharing algorithms remain widely popular and continue to serve as baselines for comparing more advanced algorithms, they are subject to two notable drawbacks – they entail sending large volumes of data over the network and they require all clients to use the same model architecture regardless of the resources they possess. The former drawback may cause significant carbon emissions [102], while the latter may lead to unfairness and in general compromise the accuracy of the model being trained [1, 2, 59]. When deploying FL algorithms on production environments, such issues are exacerbated by the inherent heterogeneity[1] present in DCML settings to such an extent [9, 69, 102], that some researchers go as far as to say that "sharing parameters to transfer knowledge *[...]* is a wrong design choice" [14].

Given the above, researchers have started reconsidering the use of a uniform ML model across all clients and have begun devising novel approaches aimed at reducing the computational demands on low-end clients during DCML training. These efforts have led to the exploration of approaches, such as Knowledge Distillation, Partial Training, and Split Learning within the DCML setting. However, to the best of our knowledge, these algorithms

[1]In DCML, by *client heterogeneity* we might consider the heterogeneity of data-generating processes (DGPs), i.e. data non-IIDness, or the heterogeneity of hardware capabilities. This paper focuses on the latter, which includes variations in clients' network connection speeds, available memory, and computing power.

have not yet been comprehensively pitted against each other. Moreover, as FL research primarily occurs in simulation environments, several such algorithms have never been deployed in a realistic heterogeneous testbed.

This paper fills this gap by providing a comprehensive review and experimental comparison of the state-of-the-art approaches designed to enable collaborative training among clients with heterogeneous computational capabilities. That is, we focus on algorithms that either support Model Customization,[2] i.e., allow different clients to use different model architectures, or reduce the burden posed on the clients by offloading part of the training computation to the server.

1.1 Paper Methodology, Structure, Contributions

To compile relevant literature for this survey, we conducted a systematic search on Google Scholar using the keywords "Hardware Heterogeneity", "Federated Learning", "Split Learning" and "Distributed Machine Learning". The search was limited to academic publications and preprints. The inclusion criteria focused on works published within the last five years,[3] referencing earlier fundamental papers when necessary. The resulting papers were screened based on their titles, abstracts, and relevance to the survey's scope.

As shown in Table 1, this is the first paper that compares a diverse set of client-heterogeneity-aware DCML algorithms both in a simulation environment and in a real-world testbed consisting of physical devices. Other relevant papers either offer a literature review [85] or, when they include experiments on real-world devices, compare only baseline FL algorithms [7, 93, 95] and, in the case of Gao et.al. [28], some split-learning algorithms.

To summarize, the main contributions of this paper are the following:

(1) We present the first survey of DCML algorithms explicitly designed to support model customization or reduce the computational requirements on low-end clients by offloading part of the training burden to the server.
(2) We propose a taxonomy of the existing algorithms in the field and discuss the relation and the similarities between them.
(3) We thoroughly compare representative algorithms from federated knowledge distillation, partial training, and split learning using not only inference accuracy but also metrics that are often neglected in research [7], e.g., network usage and CPU consumption.

We first present a thorough overview of the three main families of algorithms that meet this paper's criteria:[4] we discuss federated knowledge distillation (FKD) in Section 2, partial training (PT) in Section 3, and split learning (SL) in Section 4. We then present our experimental study, wherein we test representative algorithms from each of the groups in both a simulation environment (Section 5) and a real-world testbed (Section 6). In Section 7 we discuss the implications and limitations of our study, and we conclude the paper in Section 8. The code, which can be used to reproduce the results reported in this paper, is publicly available at https://github.com/sands-lab/flower_dcml_algorithms.

2 Model Customization via Federated Knowledge Distillation

Knowledge Distillation (KD) was initially introduced to transfer knowledge between a large "teacher" model and a smaller "student" model [12, 38] and hence enable model deployment to devices with low computational and

[2]The terminology we use, "Model Customization", should not be confused with the concept of "Model Personalization" [85], which is commonly used in FL literature to indicate algorithms that aim to improve the predictive performance by tailoring models to the characteristics of participating clients' data in cases of statistical data heterogeneity among clients.
[3]To our knowledge, hardware heterogeneity had not received attention in the research community prior to this period.
[4]To our knowledge, no other algorithm meets the paper's inclusion criteria. For example, asynchronous FL algorithms, while addressing the issue of stragglers by using stale model updates, still assume that the model can fit within the memory of every device [98]. Additionally, quantization – whether quantization-aware training to lower computing requirements [67] or gradient quantization to reduce data communication [13, 78] – is orthogonal to the approaches considered in this paper.

Table 1. Comparison of this paper with related ones.

	Consider Federated Knowledge Distillation Algorithms	Consider Partial Training Algorithms	Consider Split Learning Algorithms	Deploy on Real-World Testbed
Tan et.al. [85]	✓	✓	✓	✗
Baumgart et.al. [7]	✗	✗	✗	✓
Woisetschläger et.al. [93]	✗	✗	✗	✓
Wong et.al. [95]	✗	✗	✗	✓
Gao et. al. [28]	✗	✗	✓	✓
This paper	✓	✓	✓	✓

memory capabilities [12]. Recently, KD has been applied in DCML scenarios, leading to the emergence of the Federated KD (FKD) family of algorithms. Here, KD enables transferring knowledge between the clients and the server and vice-versa through logits rather than model parameters, which brings the following benefits:

(1) Sending logits instead of high-dimensional models reduces the volume of exchanged data.
(2) Logits provide a model-agnostic interface, allowing each client to develop a model architecture that best suits its computational and memory capabilities [52].
(3) Avoiding model parameter sharing makes the algorithms significantly more robust to adversarial attacks and less prone to privacy leakages [14].

The main challenge of applying KD algorithms in DCML settings lies in the requirement that the teacher and the student models are to be evaluated on the same data points, while in DCML clients are not supposed to share any raw data among them. We next investigate the algorithms, that have been proposed to cope with this challenge.

2.1 FKD without External Dataset

The FedKD [96] and the FML [77] algorithms propose a simple method for integrating KD in DCML settings. During the local training phase, clients train both a small globally shared model and their private models simultaneously with codistillation [5]. In other words, the model trained collectively across all clients is used to inject knowledge into clients' private models. However, while the algorithms allow clients to customize their model, they impose a substantial computational burden as clients must concurrently train two models instead of one.

Another simple way to apply KD in DCML settings is to define the teacher knowledge as a fixed set of vectors ("prototypes"), each representing a given class. For instance, in the Federated Distillation (FD) algorithm [45] the server determines the average logits for every class across all clients. Such information is used during the local training phase, in which clients penalize deviations of their outputs from the global logit of the corresponding target class. This training procedure allows for a decrease in the communication overhead by several orders of magnitude, however, the accuracy drop of such an algorithm might be as severe as 25% when compared to the plain FedAvg algorithm [45]. In place of exchanging the average logits, averaged per-class higher-dimensional intermediate embedding vectors might be exchanged instead, as in the FedProto algorithm [86]. In such a case, the increased communication cost is compensated by better inference capabilities. Note, that both the FD and the FedProto algorithm trivially allow clients to customize their models – the only requirement, in the case of the FedProto algorithm, is that all the client models need to share some embedding dimension.

The FedGKT algorithm [36] develops a model on the server without exchanging model parameters. Clients start by training a small model that is horizontally divided into an encoder and a classification head. The encoder processes raw data to generate intermediate embeddings, while the classification head – potentially comprising multiple layers – uses these embeddings to produce final predictions. After local training, clients share the intermediate embeddings, final logits, and ground-truth labels for each point in their dataset with the server. The server then uses this information to train a large model that takes the embeddings and predicts the target classes. In other words, the server trains a more complex classification head than the one used locally by clients. In FedGKT, both the client and the server employ KD to improve model convergence. Although innovative, this algorithm requires clients to reveal their distribution over labels and does not develop a fully usable model on the server – such a model lacks the encoder part, which is only available to the clients that participated in the training process.

2.2 FKD with External Dataset Dependency

The performance of the FKD approaches might be improved with an external dataset with similar properties to the underlying clients' datasets. In the Cronus algorithm [14], a public dataset is distributed to all clients, and after every training epoch clients share the logits for a subset of data points from this dataset with the server, which aggregates these values. During local training, the clients simultaneously train their local model on both their private labeled data and the public data labeled with the global logits. In the related DS-FL algorithm [44], the server aggregates the received logits with an alternative entropy-based approach, and during local training, the clients first complete the KD training stage before training the model on their local dataset. The FedMD algorithm expands on these concepts by incorporating a transfer learning phase, as client models first undergo pre-training on the public dataset. Following this warm-up phase, the algorithm is akin to DS-FL, differing only in the logit aggregation method, which in FedMD is the average.

In the algorithms just presented, the server's role primarily involves lightweight synchronization tasks. Conversely, several model-sharing algorithms leverage the server's computational capabilities to enhance model training. For instance, in the FedDF algorithm [56], multiple model architectures are concurrently trained, and knowledge transfer occurs between these architectures using KD. Specifically, during a KD training phase on the server, fine-tuned client models act as teachers, while aggregated global models are treated as students. Similarly, the Fed-ET algorithm [19] allows clients to choose among a predefined set of model architectures with a common classification head. A consensus on predictions from fine-tuned client models is used to label data in the public dataset and these pseudo-labels are then utilized to train a large server model with an identical classification head as the client models. To obtain the models that will be sent to clients in the next training iteration, Fed-ET averages the lowermost layers of the fine-tuned client models and incorporates the classification head of the server model. The related FedAUX [75] algorithm extends the FedDF algorithm by introducing the so-called *certainty scores*, which quantify the similarity of the clients' local data to the data in the global dataset and are hence used to weight the logits produced by the client models during the server KD stage. Certainty scores are obtained after training a logistic regression model that aims to differentiate between data points in the local dataset and the ones in the global dataset. Yet, the FedDF, Fed-ET, and FedAUX algorithms force the client to choose among a predefined number of model architectures and require the sharing of the full model updates.

In the FedGEMS algorithm [18] clients compute logits locally for each data point in the public dataset and transmit this information to the server. The server aggregates these predictions and uses the pseudo-labels to train a global model. Additionally, KD also occurs on the client side: selected clients receive pseudo-labels for the public dataset and, similarly to the DS-FL and FedMD algorithms, train their local models to minimize the discrepancy between their predictions and those received from the server. Extending this approach, the MHAT algorithm [42] trains the server model using a combination of known target labels and client-produced logits.

2.2.1 FKD with model personalization. All the methods discussed thus far operate under the assumption that every client receives identical information from the server, such as uniform global logits. However, this may lead to suboptimal model performance in the presence of significant statistical heterogeneity among client data. The KD-pFL algorithm [105] overcomes this limitation by introducing a square matrix that quantifies data similarity between every pair of clients. This similarity matrix is used to construct personalized logits, i.e., each client receives a weighted average of other clients' logits based on the matrix coefficients. The matrix, which expresses the similarity between clients' data, is trained on the server using KD.

The COMET algorithm [20] combines FKD algorithms with client clustering techniques [11, 31, 76] to compute personalized logits. In detail, the COMET algorithm clusters the logits sent by clients to the server using the K-Means clustering algorithm. The resulting centroids are then sent back to the clients, who select the centroid closest to their logits and use it during local training for KD.

2.3 FKD Summary

Despite the promising perspectives, FKD algorithms still face several issues and open questions:

(1) *Impact of model heterogeneity:* Though most algorithms are designed to work even in cases of extreme model heterogeneity by allowing a "continuous" space of possible models, experimentally it has been observed, that allowing some clients to use too simple models might hinder convergence of all the models [52]. While lowering the impact of the logits produced by simple models represents a potential means to ameliorate this, a thorough study of the impact of severe model heterogeneity is required.
(2) *Model architecture selection:* The analysis of how clients should independently choose their model architecture and its implications in real-world deployments has yet to be addressed.
(3) *Statefulness*: Algorithms that do not develop a fully usable server model require clients to be stateful, i.e., clients must train their model from scratch and hence be involved in model training across multiple training rounds. While feasible in cross-silo FL with a small number of reliable clients, this requirement is impractical in cross-device FL settings, where sampling from millions of devices may result in less than 1% of clients participating in training [82].
(4) *External dataset dependency:* All algorithms that develop a fully usable server model depend on an external dataset. Yet, obtaining an appropriate centralized dataset might not be feasible, and utilizing a dataset with a different distribution than those of the clients could lead to degraded model performance [56].

We conclude this chapter by reporting in Table 2 the main characteristics and properties of the algorithms we analyzed.

3 Model Customization via Partial Training

3.1 Characterization of PT approaches

Partial Training (PT) represents a group of model-sharing FL approaches that depart from the requirement of homogeneous client models by permitting each client to receive and train only a subnet of the global model. Such sub-models are obtained by dropping neurons in fully-connected layers, excluding filters in convolutional layers, and reducing the depth of the network.

In PT clients are therefore oblivious to the fact that the model they receive is merely a segment of the larger model held by the server because, as outlined in the proposed generalized Algorithm 1, the server alone manages all the coordination of the various sub-models. In particular, during each server epoch, the server selects a portion of the model for each participating client in the current training round and extracts the relevant weights. In Algorithm 1 we indicate that the task of constructing sub-models is carried out by the `Decompose` function, which takes as input the global model architecture θ^t, client meta-data κ_i^t, and possibly some algorithm-specific state γ^{t-1}. The client meta-data κ_i^t might consider the clients' computational capabilities, their available memory, and

Table 2. Summary of FKD algorithms. C is the number of classes, e the dimension of the embedding space, N and N_i the number of parameters of the global and client i's model, $k \in (0, 1)$ an arbitrary constant, and $|D|$ and $|D_i|$ the cardinality of the public and client i's private datasets respectively.

Algorithm	Required distillation dataset	Distribute public dataset to clients	Fully customized model architecture	Number of exchanged floating point values per round by client	Distillation happens on	Available server model
FedKD [96]	✗	✗	✓	N	Client	✓
FML [77]	✗	✗	✓	N	Client	✓
FD [45]	✗	✗	✓	C^2	Client	✗
FedProto [86]	✗	✗	✓	$e \cdot C$	Client	✗
FedGKT [36]	✗	✗	✓	$\|D_i\| \cdot e$	Client & Server	✗
Cronus [14]	Unlabeled	✓	✓	$k \cdot \|D\| \cdot C$	Client	✗
DS-FL [44]	Unlabeled	✓	✓	$k \cdot \|D\| \cdot C$	Client	✗
FedMD [52]	Labeled	✓	✓	$k \cdot \|D\| \cdot C$	Client	✗
FedDF [56]	Unlabeled	✗	✗	N_i	Server	✓
Fed-ET [19]	Unlabeled	✗	✗	N_i	Server	✓
FedAUX [75]	Unlabeled	✓	✗	N_i	Server	✓
FedGEMS [18]	Labeled	✓	✓	$\|D\| \cdot C$	Client & Server	✓
MHAT [42]	Labeled	✓	✓	$\|D\| \cdot C$	Client & Server	✓
KD-pFL [105]	Unlabeled	✓	✓	$\|D\| \cdot C$	Client	✗
Comet [20]	Unlabeled	✓	✓	$\|D\| \cdot C$	Client	✗

Algorithm 1 General structure of Partial Training algorithms

```
Require: $T > 0$ number of training rounds, $C$ set of training clients
1: Server initializes global model $\theta^{(1)}$
2: $\gamma^0 \leftarrow$ initialize parameters for sub-model extraction
3: for $t = 1$ to $T$ do
4:     $S \subseteq C \leftarrow$ sample subset of available clients
5:     $\{\kappa_i^t\}_{i=1}^{|S|} \leftarrow$ collect meta-information of the devices
6:     $\{\theta_i^t\}_{i=1}^{S}, \gamma^t \leftarrow \mathtt{Decompose}\big(\theta^t, \{\kappa_i^t\}_{i=1}^{|S|}, \gamma^{t-1}\big)$
7:     for $i \in S$ in parallel do
8:         Server sends model $\theta_i^t$ to client $i$
9:         $\theta_i^{t+1} \leftarrow$ train model $\theta_i^t$ on client $i$'s dataset $D_i$
10:        Client $i$ sends updated model $\theta_i^{t+1}$ to server
11:    end for
12:    $\theta^{t+1} \leftarrow \mathtt{Aggregate}\big(\{\theta_i^{t+1}\}_{i=1}^{S}, \gamma^t\big)$
13: end for
```

some properties of their local data, while γ^{t-1} might include information about the way the sub-models were constructed in the previous epochs, the current value of model parameters, and the state of the random number generator. After extracting the sub-models, the server transmits the model segments to the clients, who train the received model and upon completing the local training, send back the updated model parameters to the server.

Finally, the server aggregates the weights using its `Aggregate` function by considering how the sub-models were constructed.

The decomposition and the aggregation function therefore form a pair that characterizes any PT algorithm. In particular, depending on the properties of these two functions, PT algorithms may be categorized into static, dynamic, and independent subnetwork training (IST).

3.2 PT with Static Decomposition

In static decomposition the server constructs the sub-models deterministically and consistently across all the training rounds and clients. That is, the sub-model returned by $\texttt{Decompose}(\theta^t, \kappa_i^t, \gamma^{t-1})$ are uniquely determined by the value κ_i^t, while the state γ^{t-1} does not influence the resulting models. Note, that this does not imply that some client i necessarily receives the same sub-model in all training rounds, as κ_i^t might change over time.

Two representatives of this category are the HeteroFL [23] and the FjORD [39] algorithms. In HeteroFL the server determines a discrete number K of model capacities $\{\alpha_i\}_{i=1}^K$ and constructs the corresponding sub-model architectures $\{\theta_{\alpha_i}\}_{i=1}^K$ by varying the width of the network, that is, by changing the number of channels in convolutional layers and the number of neurons in fully-connected layers. More in detail, the submodels are constructed in such a way, that for any two model capacity classes $\alpha_1 < \alpha_2$, it holds that $\theta_{\alpha_1} \subset \theta_{\alpha_2}$, so that the models effectively form a hierarchy. Similarly, the FjORD algorithm [39] introduces a hierarchy of nested sub-models using the so-called Ordered Dropout.

In some cases, the properties of the data might be used to determine the sub-models to be formed. For instance, small datasets for NLP tasks are likely to contain only a subset of all the possible tokens, and features in the click-through rate domain are typically extremely sparse [73]. In such cases, the inherent sparsity of the data offers a natural way for sub-model construction. The FedSelect [15] algorithm leverages such data sparsity to construct the submodels, for instance, by removing the weights associated with the input neurons that always take the value 0. Unfortunately, this method is limited as it is subject to the sparsity of the data (for instance, it cannot be applied to image data). Furthermore, it can only be applied in the first and last fully connected layers.

3.3 PT with Dynamic Decomposition

A limitation of the PT algorithms with static decomposition, which directly stems from their static nature, lies in the fact that they allow the training of a model only as large as the largest model that can be trained on the clients. Dynamic decomposition algorithms avoid this model size bottleneck by iteratively serving different model parts of the model. Such iteration may be achieved by either constructing the submodels pseudo-randomly or by introducing some heuristics regarding which parts of the model should be trained. An example of the former is the Federated Dropout algorithm [13, 92], in which the server translates the meta-information κ_i^t into a single scalar $\alpha \in (0, 1)$ and extracts the submodel by randomly sampling a fraction α of neurons in fully connected layers and channels in convolutional layers. However, recent research has shown the ineffectiveness of this approach [17]. In the related FedSPU algorithm [63] the server sends the whole model to the clients, yet it instructs them to freeze a random part of the network sampled randomly. In contrast with the Federated Dropout algorithm, this solution suffers from higher memory usage and forward propagation time, while the backpropagation time remains unaltered.

Two representatives of PT with heuristics-based model extraction are the FedRolex [4] and the PriSM [62] algorithms. In the former, different model parts are trained with a rolling strategy, i.e., a rolling window iteratively loops over the entire model to extract the sub-models. Conversely, in the PriSM algorithm the server creates low-rank sub-models using the singular value decomposition. Some model pruning algorithms[5] fall into this

[5]Not all model pruning approaches fit within the proposed formulation. For instance, in the Hermes [51], Sub-FedAvg [90], and PruneFL [46] algorithms, clients initially receive and train the full model. During training clients incorporate a regularization term into the loss function,

heuristic-extraction category as well; for instance, in the FedMP [47] and the FL-PQSU [99] algorithms, the server constructs submodels by selecting the neurons with the highest importance. Such importance is defined using the *l1* norm, e.g., the importance of a filter in a convolutional layer is defined as the sum of the absolute values of the corresponding kernel's weights.

3.3.1 *PT via Independent Subnetwork Training.*

Independent subnetwork training (IST) can be viewed as a special case of PT with dynamic decomposition. In contrast to the latter, the `Decompose` function randomly creates models that have *separate and non-overlapping* segments of the model being trained [103]. This design simplifies the `Aggregate` function, rendering it trivial: if a parameter was trained in an epoch, the server copies the updated value into the new version of the global model; otherwise, the previous value of the parameter is retained.

In its original formulation, IST was proposed for fully connected layers [103]. In such a case, the sub-models are constructed by partitioning the neurons in every hidden layer into equally sized groups and extracting the weights connecting any two neurons that belong to the same partition [103]. IST has later been integrated into other model architectures. The ResIST algorithm applies the IST principles for training the ResNet architectures [37] by distributing residual blocks to clients in such a way that each client trains a shallower network. Next, the GIST algorithm [94] applies IST to graph neural networks and AsyncDrop[25] to convolutional neural networks.

IST has also been applied in hierarchical FL settings (HFL).[6] Namely, the HIST algorithm [27] proposes to distribute an independent subnetwork to every edge server and make each such server train its model segment with a model-sharing FL algorithm for a given number of server rounds, before aggregating the results on the global server level and distributing new subnetworks to the edge servers.

IST speeds up the training convergence if compared to local SGD [84] and data parallel training (DPT), though such speed-ups typically come at the cost of a slightly reduced inference accuracy [26, 27, 94, 103]. In contrast with the DPT, wherein at some point the training time starts increasing as we add more training machines because of the gradient communication overhead overshadowing the benefits of parallel computation [26, 103], IST does not suffer from performance degradation issues.

Despite its sound theoretical background [79], IST did not gain much attention in DCML settings. This has led to a notable research gap concerning various practical deployment conditions. For example, the blind replication of client-provided values by the server in the updated model introduces vulnerabilities to potential adversarial attacks. Additionally, the influence of statistical data heterogeneity on the efficacy of IST-trained models remains largely unexplored.

3.4 Partially Local Federated Training

The Partially Local Federated Training (PLFT) algorithms offer a compromise between PT and the SL approaches we discuss in Section 4. Similar to SL, PLFT involves partitioning the model horizontally into public and private segments. Conversely, akin to PT, these algorithms update the clients' models by sharing a portion of the overall model parameters. Specifically, the public segment of the model – comprising either the lowermost or uppermost layers [70] – is trained using a model-sharing algorithm, while the remaining layers remain private to each client.

In general, deciding which model parameters to designate as shared and which ones as private is an open question, as different choices perform best in different scenarios [36]. In the LgFedAvg [54] and the FedGH [101] algorithms, the lower layers in the model are private and the upper ones are shared, while in the PerFed algorithm [6] the opposite route is taken, i.e. the lower layers are collaboratively trained and the upper ones are privately trained by the clients.

which encourages certain model parameters to be pushed to zero, effectively reducing the model size. We do not consider these algorithms as they require the clients to train the whole (large) model before obtaining a smaller version.

[6]In HFL clients only communicate with the geographically closest edge server and edge servers communicate with the global server. Therefore, there is a hierarchy in the communication.

These algorithms reduce the communication burden if compared to algorithms that require the exchange of the full model, however, they require the clients to be stateful. The FedRecon algorithm [82] attempts to solve this issue by introducing the notion of a “Reconstruction” algorithm, which is used to initialize the private model weights as the clients need them. More in detail, a client selected for training partitions its private dataset into a *support* and a *query* part, and uses the former to initialize the private parameters and the latter to train the whole model after the private part has been initialized.

PLFT algorithms can handle heterogeneous models under the condition, that the customized client models have the same architecture of the shared layers. However, to the best of our knowledge, this direction has never been empirically tested. We explore this direction in Section 5.

We conclude this section by listing in Table 3 the main characteristics of the algorithms presented in this section.

Table 3. Summary of Partial Training Algorithms.

	Decomposition type	Client receive non-overlapping model weights	Deterministic weight extraction	Model partitioning
HeteroFL [23]	Static	✗	✓	Vertical
FjORD [39]	Static	✗	✓	Vertical
FedSelect [15]	Static	✗	✓	Vertical
Federated Dropout [13]	Dynamic	✗	✗	Vertical
FedRolex [4]	Dynamic	✗	✓	Vertical
FedSPU [63]	Dynamic	✗	✗	Vertical
PriSM [62]	Dynamic	✗	✓	Vertical
FedMP [47]	Dynamic	✗	✓	Vertical
FL-PQSU [99]	Dynamic	✗	✓	Vertical
IST [103]	Dynamic	✓	✗	Vertical
ResIST [37]	Dynamic	✗	✗	Vertical
GIST [94]	Dynamic	✓	✗	Vertical
HIST [27]	Dynamic	✗	✗	Vertical
AsyncDrop [25]	Dynamic	✓	✗	Vertical
LgFedAvg [54]	Dynamic	✗	✓	Horizontal
FedGH [101]	Dynamic	✗	✓	Horizontal
PerFed [6]	Dynamic	✗	✓	Horizontal
FedRecon [82]	Dynamic	✗	✓	Horizontal

4 Split Learning

4.1 Introduction to Split Learning

Split learning (SL) [34, 71, 91] is a distributed model training approach where the model is horizontally partitioned into multiple segments distributed across two or more training nodes. In its simplest form, SL involves only two nodes, i.e., the client possessing the training data and the first k layers of the model, and the server possessing the remaining $N - k$ layers, N being the total number of layers in the model. We visually compare the difference between SL, PT, and traditional model-sharing algorithms in Figure 1.

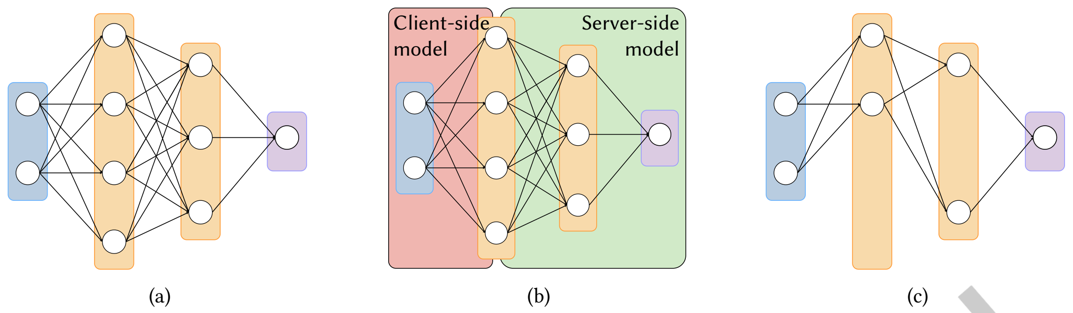

Fig. 1. (a) In model-sharing algorithms, the whole model is distributed to all the training clients. (b) In SL the model is partitioned horizontally. (c) In PT the model is partitioned vertically, i.e., clients train a reduced yet complete version of the model.

After both actors initialize their model segment, training proceeds through standard backpropagation. Specifically, the client first conducts a forward pass on a batch of local data on its model segment and sends the resulting so-called "smashed" embeddings along with the corresponding target labels to the server. The server continues the forward pass on its model portion, determines the loss, and updates its weights by backpropagating the error until its first layer. Then, the server communicates the gradient information to the client, enabling this way the client to perform the backward pass on its segment of the neural network.[7] Therefore, SL allows clients to train a larger model than the one they could on their own and without sharing their raw data.

The above two-node setting can be extended with several architectural variations. Of particular interest is the so-called "U-shaped" architecture [66, 91, 106], in which the client holds both the lowermost and topmost layers of the model. On the one hand, this configuration removes the need for the client to share target values with the server, as the client computes the loss itself, but on the negative side, requires four passes over the network – two during the forward pass and an equal number during the backward pass.

When multiple clients aim to collaboratively train a model, several approaches are available. Gupta et al. propose a formulation where clients take turns in training. That is, after client i completes training, it sends the updated client-side model parameters to client $i + 1$ in a round-robin fashion [34]. However, given that only one client is engaged in model training at any given time, this leads to low resource usage, increased convergence time, and, when clients' data present statistical heterogeneity, might lead to the so-called "catastrophic forgetting" [24, 89].

To overcome the low resource utilization issue, the SplitFed algorithm [87] proposes to combine SL with model-sharing algorithms by training the client-side models in parallel and periodically averaging them as in the FedAvg algorithm. The SplitFed algorithm comes in two variants, as the authors observe that during training the server model might either be private to every client (SplitFed $v1$) or be shared among all clients (SplitFed $v2$). When each client trains its private server model, such server models are averaged at the end of the training round. Consequently, this version is effectively equivalent to the FedAvg algorithm. In the SFLG algorithm [29] authors generalize these algorithms by observing that the number of models being trained on the server may be any value between one and the number of clients. The SplitFed $v3$ algorithm [30] proposes to keep the client-side part of the model private to clients and only average the updates of the server model to reduce catastrophic forgetting [49]. On the negative side, this prevents from developing of a public client-side mode that may be served to new clients.

[7]Such a two-node scenario is, from a technical point of view, equivalent to pipeline-parallel training [43].

One of the main design questions in SL is determining the number of layers to be trained on the client. Employing a shallow network on the clients reduces their resource usage; however, it also increases the risk of potential privacy breaches [3]. Kim et. al. propose an algorithm that determines the optimal number of layers to be trained on clients in cases of IoT devices by introducing a set of utility functions, which take into consideration the energy consumption, the privacy of the data, and the consumed time [48].

Compared to model-sharing FL, SL reduces the volume of shared data over the network when training large models and in the presence of a large number of clients [81]. Conversely, SL has the disadvantage of requiring network involvement for every batch of data. To address this network issue, Han et al. propose the FSL algorithm, where clients use a shallow auxiliary head to compute a loss for updating the client-side model [55]. The loss computed on the server updates only the server-side model, removing the need for the server to share gradients with clients. Similarly, in the AdaSplit algorithm [21], client models are trained using locally generated losses. However, unlike the FSL algorithm, in AdaSplit clients first train their models for a fixed number of rounds and subsequently update the server model with sparse updates. Therefore, the purpose of the client loss in FSL and AdaSplit is to enable updating client models in the absence of the gradient from the server. However, client losses can also be used to maximize the amount of information contained in the smashed data as in the LocFedMix algorithm [64].

4.2 Split Learning Summary

In summary, SL represents a distinct group of DCML algorithms separate from the FL category. Broadly, SL algorithms facilitate collaborative model training among clients with minimal memory and computational requirements by delegating some computational tasks to the server. The presented analysis evidenced that SL needs further research in terms of:

(1) *Scalability*: as every client consumes some server resources, such as memory and computing time, SL approaches seem best suited to small-scale scenarios. Consequently, there is a need for a study to address its scalability issues and propose effective mitigation strategies.
(2) *Privacy*: an issue in SL arises from clients sharing target labels and the possibility that the server may reconstruct raw data based solely on the embeddings. While the former issue can be addressed with the U-shaped architecture, the latter remains an active area of research [68].
(3) *Implementation*: to the best of our knowledge, no existing framework is specifically designed for easy prototyping, implementing, and deploying SL algorithms. We develop such a framework as part of the contributions of this paper, see Appendix A.

We conclude by reporting the properties of the considered SL algorithms in Table 4.

Table 4. Summary of Split Learning algorithms.

	Client avoids sharing target label	Client collaboration	Number of server models g during training
SL [34]	✗	Sequential	$g = 1$
U-Shaped SL [91]	✓	Sequential	$g = 1$
SplitFed $v1$ [87]	✗	Parallel	$g = n$
SplitFed $v2$ [87]	✗	Parallel	$g = 1$
SplitFed $v3$ [30]	✗	Parallel	$g = 1$
SFLG [29]	✗	Parallel	$1 \leq g \leq n$
FSL [55]	✗	Parallel	$g = 1$
AdaSplit [21]	✗	Parallel	$g = 1$

5 Experimental Analysis in Simulation Environment

In this section, we conduct a comprehensive evaluation of a diverse set of algorithms discussed in this paper. Our objective is to provide insights and contributions as follows:

- To chart the landscape of algorithmic effectiveness, we analyze the performance of the implemented algorithms across various metrics such as model accuracy, convergence speed, communication overhead, and resource utilization.
- We validate the reported results from the original papers and uncover new insights by executing the algorithms in a real-world testbed.
- We provide actionable guidelines that can assist researchers and practitioners in navigating the trade-offs associated with the analyzed algorithms and hence help them make informed decisions regarding the most suitable collaborative algorithm for their specific use case.

5.1 Experimental setup

Scenario: We consider a scenario where a group of devices with varying computational capacities collaborates to train a shared model. These devices are grouped into k clusters based on their computational capabilities, i.e., all clients within a cluster can train a common model architecture. We consider $k = 3$, distinguishing between:

(1) High-capacity devices, which can train a *Large* model.
(2) Mid-capacity devices, which can train a *Medium* model.
(3) Low-capacity devices, which can only train a *Small* model.

Unless stated otherwise, we consider 21 clients evenly distributed across the three computational tiers. Each client trains the largest model it can handle; for example, a mid-capacity device always trains a *Medium* model, even though it could also train a *Small* model. Additionally, every client uses the same model architecture across all algorithms. For instance, low-capacity devices consistently train the *Small* model across all algorithms.

Considered algorithms: The selection of algorithms for implementation is made to maximize diversity. Specifically, we choose a set of algorithms that cover a broad range of algorithmic ideas surveyed in this paper. In selecting the algorithms for the experimental section, we consider their simplicity, the clarity of their descriptions in the original papers, their novelty, and the initial results we obtain. We also prioritize algorithms with open-source implementations and those for which we can reproduce the results reported in the papers. For the FKD family of algorithms, we consider the FD [45], FedKD [96], FedMD [52], and FedDF [56] algorithms. For the PT family of approaches, we consider the HeteroFL [23], Federated Dropout [13], and LgFedAvg [54] algorithms.

Finally, we take SplitFed $v1$ and SplitFed $v2$ [87] as representative of SL algorithms. A detailed discussion of the implementation of these algorithms for our use case is provided in Appendix A.1.

We refer to algorithms that require clients to share the weights of the model being trained and hence develop a server-side model as "stateless". Examples of such algorithms include FedAvg, HeteroFL, FedDF, and Federated Dropout. Conversely, algorithms that develop a private model without sharing the corresponding model parameters, such as FedMD, FedKD, FD, and LgFedAvg, are classified as "stateful".

Experimental procedure: Our experimental evaluation consists of two stages. In the first stage, we perform a grid search over possible hyperparameter values for each algorithm to determine the best hyperparameter configuration. These initial tests take place in a simulation environment wherein we artificially mimic computational heterogeneity among (virtual) clients. Subsequently, in the second stage, we use the hyperparameter configuration determined in the first step and deploy the algorithms, depending on the experiment type, either in a simulated environment or in a real-world testbed. We limit ourselves to small-scale scenarios because of the finite number of physical devices in the testbed at our disposal. We also assume full client participation in every server round as otherwise it would not be possible to compare stateful and stateless algorithms.[8] Details about the experimental setup, including model architecture, hyperparameter search space, and data partitioning procedure, are discussed in Appendix A.

Training task: we focus on common classification benchmarks using the CIFAR10, CIFAR100 [50], and CINIC10 [22] image classification datasets and the Ag-News text classification dataset [107]. For brevity, we here present results for CIFAR10 and leave the results on CIFAR100 and CINIC10 for Appendix B and on Ag-News to Appendix C. We explore two data scenarios: *a)* data distributed in an IID fashion, where clients have similar distributions over the target labels, and *b)* in a non-IID manner, where labels are distributed based on the Dirichlet distribution [41, 104]. Regardless of the distribution of client labels, clients' datasets are divided into disjoint training, validation, and test subsets. During training, the server instructs clients to compute the validation accuracy of the current model every five server training rounds. The validation accuracy serves two purposes. First, it allows the server to detect convergence defined as the lack of improvement of the average validation accuracy over four consecutive evaluation rounds (which span $4 \cdot 5 = 20$ training rounds). Second, clients use the validation accuracy to select the version of the model that will be used at the end of training – note that each client might choose a model from a different server round. Thus, by "test accuracy" we denote the highest validation accuracy model's performance on the client's test set.

5.2 Baseline accuracy comparison

We introduce an intuitive baseline that is founded on the FedAvg algorithm. In place of training a model across *all* clients as is the case of the FedAvg algorithm, we train a model with FedAvg only across clients of the same computational capacity. That is, after clustering clients into k clusters by considering their available computing resources, we run the FedAvg algorithm k times, each time only involving clients of the considered cluster. To the best of our knowledge, no paper to date has examined this option for a heterogeneous model environment. Note, that this approach is tightly related to clustered FL algorithms [11, 76], wherein after clustering the clients according to the properties of their data, a separate model is trained independently for each cluster. In contrast with these solutions, however, in our case we introduce *architectural* model heterogeneity. Though this solution is likely to yield unsatisfactory results for clients in underrepresented capacity groups, we argue that it is a valid baseline as it entails no overhead for clients and introduces only minimal overhead for the server, as in this scenario the server is required to average and store k models in memory instead of one. We refer to this extended version of the FedAvg algorithm as *cFedAvg*.

[8]In stateless algorithms the global model is trained in every epoch, while in stateful algorithms the client model is trained only if the client is selected in the current training round.

Table 5. Test accuracy (in %) and corresponding standard error achieved by FedAvg on the proposed CIFAR10 setting w.r.t. the number of local training epochs performed on the clients. *All clients* denotes training the model across all clients, while *Cluster clients* denotes training the model across the clients in the corresponding capacity group. That is, the accuracy of the baseline cFedAvg algorithm is the weighted average of the *Cluster clients* accuracy.

	Small model		*Medium* model		*Large* model	
	All clients	Cluster clients	All clients	Cluster clients	All clients	Cluster clients
1 local epoch	76.4 ±0.4	70.7 ±1.0	82.0 ±0.2	77.1 ±0.4	83.7 ±0.2	77.7 ±0.4
2 local epochs	78.5 ±0.2	70.6 ±0.7	82.9 ±0.1	76.8 ±0.3	84.1 ±0.2	77.6 ±0.4
4 local epochs	78.3 ±0.2	70.2 ±0.9	82.7 ±0.1	76.2 ±0.5	83.7 ±0.1	77.2 ±0.4

Takeaway 1: *A trade-off exists between the size of the model and its accessibility to low-end devices, with both factors significantly impacting the final model's accuracy.*

We report in Table 5 the accuracy for the proposed baseline, wherein we train a model only across the clients that support it, as well as the accuracy we would obtain if all clients trained the same model architecture with the FedAvg algorithm.[9] The results are expected, as larger models consistently outperform smaller models and at the same time, when a given model is trained across all clients, it yields superior performance than when trained on a subset of clients. A point worth emphasizing is that the accuracy of the *Small* model trained across all clients is similar and in some cases smaller than the accuracy of the *Large* model trained only on the subset of clients that support such a model. We argue that these results collectively showcase the *necessity of employing heterogeneous model architectures in heterogeneous environments*. Ideally, algorithms should allow transferring knowledge between different capacity groups of clients. Hence, when using an ideal algorithm, the accuracy of any model should approach the accuracy that the model achieves when trained across all clients.

5.3 Accuracy with respect to dataset size

In Figure 2 we report the accuracy we obtain with the considered algorithms w.r.t. the size of the training dataset of the clients. In each experiment, every client samples and uses P data points from its training set. Afterward, the server executes the algorithm until convergence.

The SplitFed algorithms achieve the highest accuracy for any P. This is expected, as in these algorithms all clients collectively train a slightly modified version of the *Large* model – recall, that the SplitFed $v1$ algorithm is equivalent to the FedAvg algorithm with the difference, that the burden of training is distributed between clients and server. Therefore, the highest accuracy we get in the case of SplitFed $v1$, 83.3% when $P = 2000$, is in line with the results reported in Table 5. A point worth emphasizing is that in the SplitFed algorithms, every client holds and trains 57184 parameters, which is very close to the number of parameters of the *Small* model (52823 parameters) used by low-capacity devices in FL algorithms. It follows, that all the devices, including the low-capacity ones, can train the same *Large* model.

Takeaway 2: *In an IID data scenario, SL consistently yields the best accuracy.*

[9]These results were obtained in the simulation environment, allowing for the emulation of scenarios where all clients possess the capability to train the largest model. However, according to the problem definition, low-capacity clients cannot use the *Large* model.

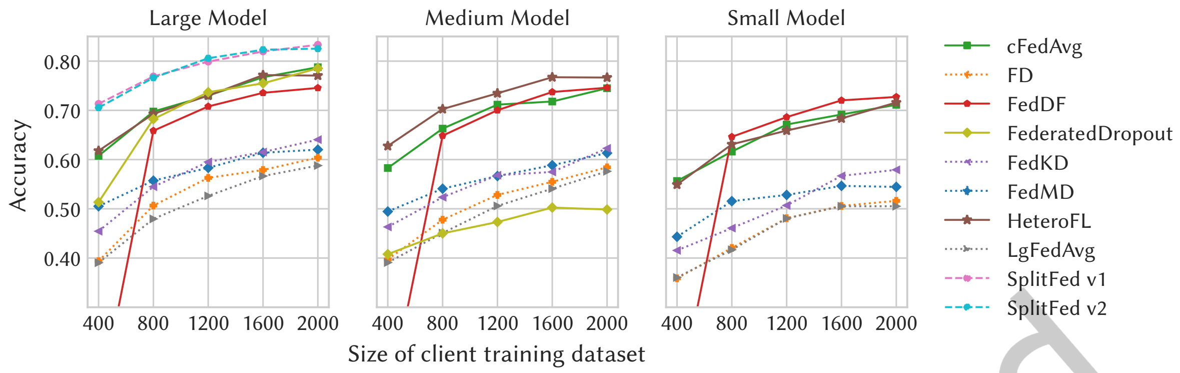

Fig. 2. Average client test accuracy w.r.t. the training dataset size and model size. Each subplot shows the average test accuracy of clients training the corresponding model, e.g., the accuracy in the "Large model" plot is the average test accuracy of the clients that train the *Large* model. In this and the following plots, solid lines indicate the stateless FL algorithms, dotted lines stateful FL algorithms, and dashed lines SL algorithms.

Within the family of stateless FL algorithms, the HeteroFL algorithm is particularly successful for the mid-capacity cluster of devices training the *Medium* model, while the performance is very similar to the cFedAvg algorithm for the *Large* and *Small* models.[10] Regarding FedDF, we notice that the algorithm requires the client models to be very well fine-tuned on clients' datasets for KD on the server to be useful. In the opposite case, KD might be detrimental – for instance, when the training set size consists of only 400 instances, the algorithm does not converge. Apart from this, the FedDF algorithm helps the *Small* model to boost its performance, as the FedDF algorithm achieves the highest test accuracy for the *Small* model for any $P \geq 800$.

We also implement and test the Federated Dropout algorithm but we obtain unsatisfactory results, most notably for the *Small* model – the average accuracy does not reach 30% and is hence not visible in the rightmost plot in Figure 2. We hypothesize that this result is due to two reasons. First, the algorithm seems not well suited for cases in which different model sizes are trained at the same time – in [13] the experiments are performed by using a constant sub-model size for all clients. Second, the *Small* model has, using the terminology from [13], a federated dropout rate of 0.2. Such a rate is significantly smaller than the smallest dropout rate considered in the referenced paper, i.e. 0.5, which has already been shown to yield unsatisfactory results [13].

Regarding stateful algorithms, the FedMD algorithm is particularly successful when the P is small. This is due to the fact, that the importance of the public dataset consisting in our case of 1000 data points is more pronounced when there is a paucity of local training data. Conversely, the FedKD algorithm is particularly successful when the dataset is large, with the low-capacity cluster of devices having the highest benefits.

The accuracy obtained with LgFedAvg is lower than that of the other stateful algorithms. In the official GitHub repository of the project [57] it is stated, that to obtain the results shown in the paper, it is first necessary to pretrain the model with the FedAvg algorithm. We therefore hypothesize that the algorithm might be better suited for refining the model obtained with FedAvg rather than training the model from scratch.

Takeaway 3: *In an IID data scenario, there is a significant gap between stateful and stateless algorithms.*

[10]The reduction in training data introduced by the cFedAvg algorithm may result in significantly worse performance in other scenarios, such as when training a BERT model for text classification tasks. See Appendix C.

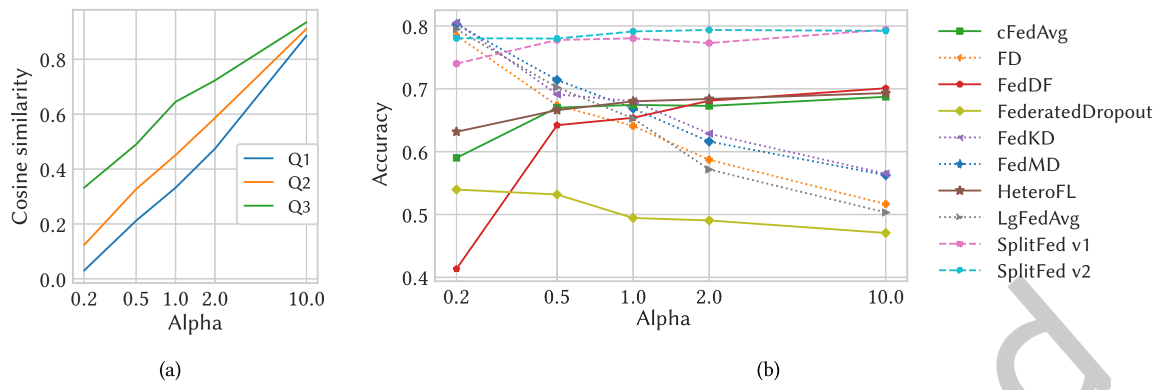

Fig. 3. (a) First, second, and third quartile of the $\binom{N}{2}$ cosine similarities between client distribution vectors, where N is the number of clients. (b) Average test accuracy across all clients obtained by the algorithms using the corresponding non-IID datasets.

5.4 Impact of data heterogeneity

We next analyze how the performance of the considered algorithms changes as the degree of data non-IIDness between clients' datasets varies. We follow a well-established practice of simulating heterogeneity by sampling data according to the Dirichlet distribution, effectively mimicking the label skew type of non-IIDness [41, 104]. The Dirichlet distribution has a parameter α, which, as shown in Figure 3(a), determines the degree of non-IIDness: the lower the value of this parameter, the more skewed the data is among the clients; conversely, as α increases, the distribution over the target labels becomes more uniformly distributed among the clients.

In Figure 3(b) we observe, that stateful algorithms perform better in cases of significant data non-IIDness. This result is caused by the fact, that when the data is highly skewed, each client has only a fraction of the overall labels in its private datasets. Consequently, the local model's task is simplified as the model needs to discriminate between fewer classes. On the other hand, as noted by several researchers [40, 108], the performance of model-sharing algorithms tends to improve as the degree of non-IIDness decreases.

Takeaway 4: *Stateful algorithms better cope with the label skew type of non-IIDness.*

5.5 Impact of client capacity

The experiments conducted thus far have assumed an equal distribution of devices across the three capacity classes. However, this scenario is unlikely to occur in practice, as certain capacity tiers are likely to be overrepresented compared to others. Here, we present how various algorithms perform as the distribution over the available capacity tiers changes. For simplicity, we assume that the 20 clients with IID data comprising this experiment may choose only among two separate models: the *Large* model and a comparison model, which can be either the *Medium* or the *Small* model.

It can be observed in Figure 4 that stateful algorithms exhibit relatively stable performance as the distribution over trained model sizes changes. However, they tend to achieve better accuracy as the percentage of clients training the *Large* model increases. This trend is most noticeable when the comparison model is the *Small* one.

Stateless algorithms are sensitive to variations in the distribution over trained model sizes. When all devices train the *Large* model, the HeteroFL, cFedAvg, and Federated Dropout algorithms yield the same accuracy.

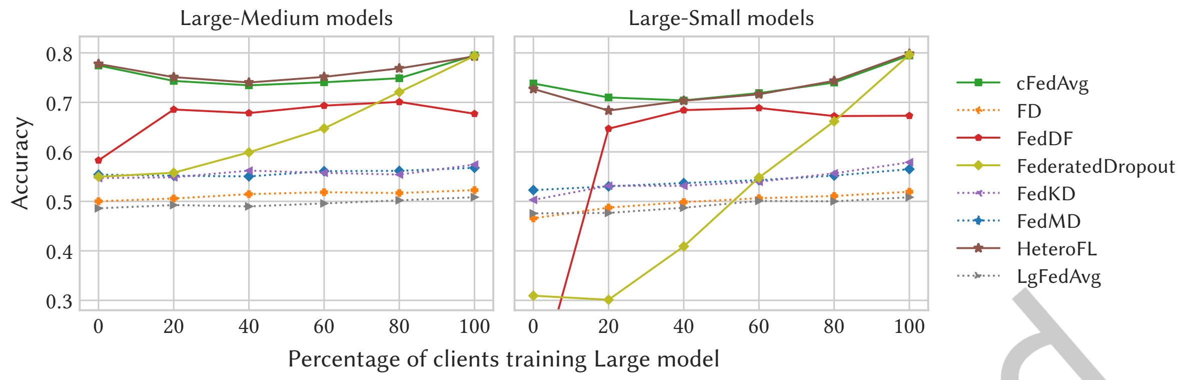

Fig. 4. Average test accuracy across all clients w.r.t. the number of devices that train the *Large* model. Clients, that do not train the *Large* model, use the *Medium* model (left figure) or the *Small* model (right figure).

However, as the percentage of clients training a *Large* model decreases, the average accuracy also decreases. For instance, Federated Dropout shows a rapid decrease in performance, especially when the comparison model is *Small*. In contrast, the cFedAvg and HeteroFL algorithms demonstrate similar performance, with HeteroFL slightly outperforming when the comparison model is the *Medium* one. Note also, that when all devices train a *Small* model, the average accuracy of the HeteroFL model is lower than that of cFedAvg, which may seem counterintuitive as both algorithms train the same model and use the same aggregation strategy. This difference arises due to the scaling of model outputs in the HeteroFL algorithm, i.e., the two algorithms have different training procedures on the clients.[11] Finally, note that we did not report the results for SL algorithms, as in the discussed scenario all the clients train the same model regardless of their capacity.

In Figure 4 we also note, that the average accuracy for both HeteroFL and cFedAvg is higher when all the devices train a smaller model if compared to the case when only a small portion of clients train the *Large* model. This demonstrates the trade-off between having multiple model sizes on one hand and training each such model only on a fraction of the clients on the other.

Takeaway 5: *In stateless algorithms, having only a marginal fraction of clients with large models might be detrimental – better let all the clients use a smaller model.*

6 Deploying the Algorithms in a Real-World Testbed

We now transition from experimenting in a simulated environment to deploying the algorithms on the Collaborative Learning Experimentation Testbed (CoLExT) [10], which includes 14 heterogeneous devices. We report the properties of the devices in the testbed in Table 6, while a picture of the testbed can be seen in Figure 5. To categorize each device into its respective capacity tier, we performed manual profiling of their performance. This involved running all models on all device types and selecting the most suitable model for each device based on training time.

Throughout this section, we train a model on an IID data setting using the CIFAR10 dataset. We set each client to use a training dataset with 1000 images to enable resource consumption comparison across different clients, and we set all algorithms to train the model for two epochs on the clients in each server training round. We apply no modifications to the FL and the SplitFed *v*2 algorithms, while for the SplitFed *v*1 algorithm we test the original

[11]We cannot make the two training procedures the same as the initial magnitudes of the model parameters are different.

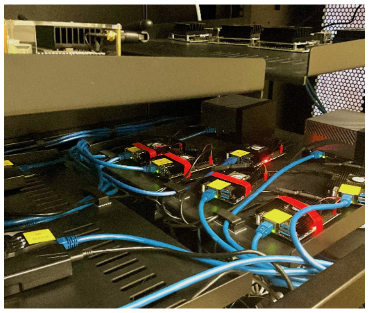
Fig. 5. Picture of the employed testbed.

Table 6. Listing of devices comprising the testbed with corresponding properties.

	RAM (GB)	CPU (cores)	GPU	Assigned capacity tier	Count
Jetson AGX Orin	64	12@2.2GHz	✓	High	1
Jetson Orin Nano	8	6@1.5GHz	✓	High	3
LattePanda Delta 3	8	4@2.9GHz	✗	Medium	4
Orange Pi 5B	16	8@2.4GHz	✗	Low	6

formulation [87] and an extended version of the algorithm that employs the U-shaped architecture [91]. We refer to the former as “Plain SL” and to the latter as “U-shaped”.

6.1 Convergence times

To begin, we examine how accuracy evolves over time. Each algorithm we analyze employs distinct training procedures and loss functions, resulting in varying training times across devices.

The results, reported in Figure 6, illustrate that the SL algorithms achieve the highest validation accuracy and converge reasonably fast. The accuracy result is in line with the discussion of Section 5.2, however in this case, the difference in accuracy between FL and SL algorithms is further amplified because in our testbed the low-capacity tier is over-represented. The difference in accuracy between the two SplitFed $v1$ versions we see in the plot is attributed to the longer training time of the U-shaped configuration. We also observe that SplitFed $v2$, where clients train a common model on the server, achieves significantly faster convergence, though some authors observe that the faster training pace of the server model may harm in some cases model convergence [65].

Regarding the FL algorithms, cFedAvg exhibits the fastest convergence rate. This rapid convergence is attributed mostly to the fact, that the training processes of different capacity tiers are decoupled. It follows, that the high-capacity cluster converges extremely quickly. However, ultimately, HeteroFL achieves the highest validation accuracy. The FedDF algorithm suffers from very slow convergence because of the expensive KD stage happening on the server. Among the stateful algorithms, the FedKD and FedMD algorithms achieve comparable accuracy, with the FedKD algorithm demonstrating faster convergence due to the fact, that it does not require training the model on the public dataset. Finally, we observe that also in this case the FD and the LgFedAvg algorithms attain the lowest accuracy among the considered algorithms.

6.2 Network consumption

Before delving into the results concerning network usage, it is important to note, that while the considered algorithms vary significantly, the types of exchanged data remain limited. That is, stateless algorithms transmit the complete model and SL algorithms transmit embeddings and gradients. Most variability in terms of transmitted data can be observed in the stateful family of algorithms, wherein clients transmit either a reduced model version (FedKD), logits on the public dataset (FedMD), or class prototypes (FD).

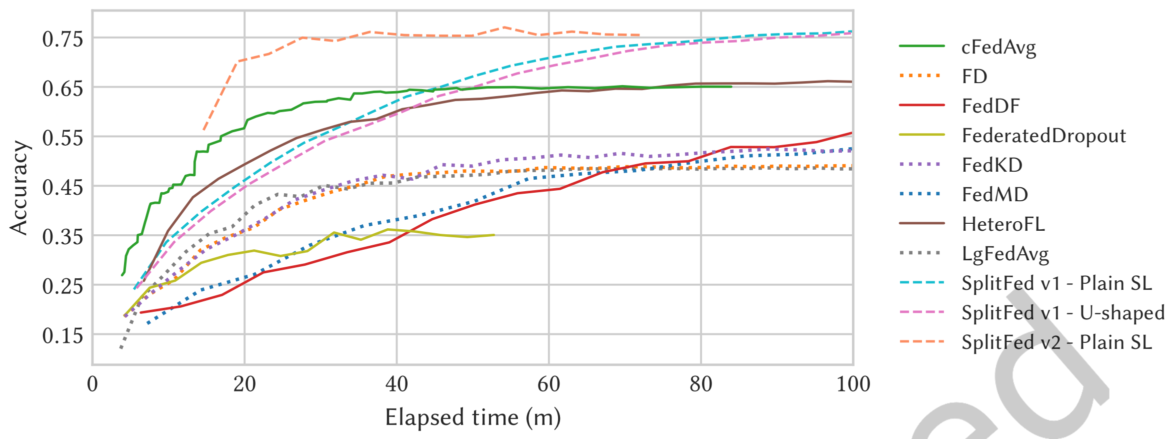

Fig. 6. Evolution of the average validation accuracy across all clients through time on the testbed.

We present in Figure 7 the volume of sent[12] and received bytes by the considered algorithms. As also noted by Gao et. al. [28], SL algorithms cause significantly higher network involvement than FL algorithms – we here do not differentiate between SplitFed *v*1 and SplitFed *v*2 because they transfer the same amount of data over the network. This fact comes as no surprise as the embedding of a single image consists in our case of 24576 parameters. It follows, that in each epoch every client, which in this experiment has 1000 data points and performs two local training epochs, sends $24576 \cdot 1000 \cdot 2 \approx 49.1$M parameters to the server. Conversely, among FL algorithms, the largest communication happens when clients send the *Large* model consisting of 1.2M parameters at the end of the training stage.

Among the FL family of algorithms, stateless algorithms exhibit traffic volumes that are one or more orders of magnitude higher than those of the FedKD and FD algorithms, a difference that gets amplified with larger model sizes. As for the FedMD algorithm, there is a considerable initial downlink investment due to clients downloading the public dataset, while the amount of data sent by this algorithm remains comparable to other stateful algorithms.

6.3 Local training resource consumption

We next analyze resource utilization during local training on clients, specifically focusing on CPU and memory usage.[13] Similar to our findings on network activity, we observe limited variability in local training procedures. We hence collectively refer to the most straightforward training method, wherein clients locally optimize the plain cross-entropy loss and is employed by the FedAvg, HeteroFL, LgFedAvg, and Federated Dropout algorithms, as "vanilla training". Conversely, we denote the regularized training procedure used by FedDF as "FedProx" [53].

We detail the training time for the first training round along with the subsequent rounds' average training times in Table 7. Notably, the initial round on devices equipped with a GPU is 68% to 77% slower than subsequent rounds due to the CUDA context creation overhead, which includes loading the driver and kernels. This initial sluggishness results in the Latte Panda, which lacks a GPU, achieving a training time comparable to the Jetson Orin Nano equipped with a GPU during the first round. Consequently, it is beneficial to utilize the GPU-equipped

[12]Sent bytes include resource consumption statistics, which are not typically sent in production environments. This traffic accounts for approximately 50kB per minute.
[13]For memory, we measure the Resident Set Size (RSS).

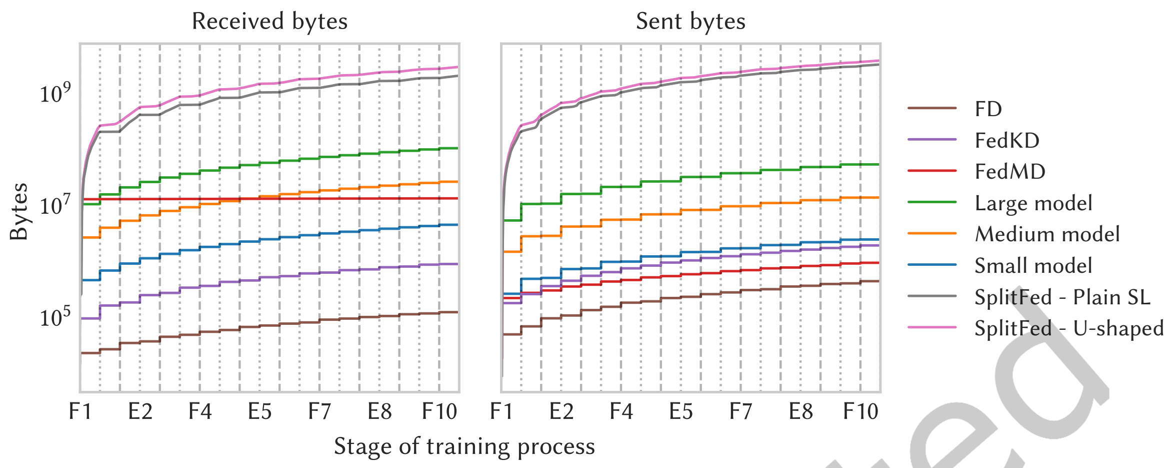

Fig. 7. Cumulative amount of sent and received bytes through training. Vertical dashed lines denote the beginning of a training round, and vertical dotted lines denote the beginning of an evaluation round. Fx represent fit round x while Ex represent evaluation round x.

Table 7. Training time in seconds in the first and subsequent epochs when training a *Small* model with vanilla training.

device type	First epoch [s]	Other epochs [s]
Jetson AGX Orin	7.87	4.67
Jetson OrinNano	9.39	5.29
Latte Panda Delta3	9.98	10.39
Orange Pi5B	19.02	17.46

Table 8. Ratio between metrics collected when training a *Large* and *Small* model with vanilla training.

device type	CPU util.	Memory util.	Round time
Jetson AGX Orin	0.898	1.017	1.214
Jetson Orin Nano	0.910	1.019	1.186
Latte Panda Delta3	1.044	1.321	10.226
Orange Pi5B	0.972	1.413	5.582

devices multiple times to amortize the initial GPU setup cost. Because of this consideration, all the values we report from this point on are measured by excluding the first training round.

Takeaway 6: *The presence of a GPU is the primary indicator of the client capacity, however, GPU initialization is slow.*

Next, we report in Table 8 the ratio between the resource usage when devices train a *Large* compared to the *Small* model with vanilla training. We note, that switching the model being trained from *Small* to *Large* causes the training time to increase by $18-21\%$ when the device has a GPU, while CPU-only devices present a time increase of multiple folds. It is also interesting to note, that memory usage does not scale proportionally to the model size because the majority of the memory consumption is not caused by the models per se, but rather by

Table 9. Ratio between memory usage and client training time when training the *Small* model with the displayed training procedure if compared to the plain vanilla training.

	FedProx		FedKD		FD	
device type	mem util	round time	mem util	round time	mem util	round time
JetsonAGXOrin	1.000	1.170	0.998	1.376	1.002	1.237
JetsonOrinNano	0.997	1.163	0.997	1.392	0.998	1.201
LattePandaDelta3	1.014	1.005	1.004	1.290	0.991	1.004
OrangePi5B	1.030	1.020	1.004	1.486	1.001	1.010

the deep learning library.[14] We therefore conclude that the size of the model, when it falls within the range of several million parameters, is unlikely to pose a memory issue. For instance, a model with 1 million parameters, assuming 32-bit precision, would occupy approximately 4MB of memory. During training, this value increases due to the storage of activations and possibly optimizer states. However, even with these additions, the memory footprint remains modest compared to that of the PyTorch library. Based on our tests, loading PyTorch into memory requires, depending on the CPU architecture (ARM or x86) between 200MB to 350MB.

Takeaway 7: *When the model being trained has a parameter count in the order of millions, the memory footprint of the model is negligible if compared to the memory occupied by the deep learning library.*

6.3.1 Comparison of FL algorithms. We report in Table 9 the ratio between different training procedures and vanilla training. We see, that as observed also by Baumgart et. al. [7], devices featuring a GPU exhibit a greater increase in training time compared to CPU-only devices in case of the FedProx and FD training procedures: for GPU-equipped devices, the round time increases range from 16% to 23%, while devices equipped solely with a CPU experience a round time increase of no more than 2%. The higher increase in training times on GPU-equipped devices is due to operations in the training procedures that cannot be efficiently performed on the GPU, e.g. computing the prototype matrix in the FD algorithm. As expected, since the FedKD algorithm trains two separate models with codistillation, the training time increases for this procedure are higher compared to the other two training procedures, with time increases ranging from 29% to 49%. Consistently with the results discussed above, also in this case we note that memory consumption does not exhibit any significant variation between the different algorithms.

6.3.2 Comparison of FL with SL algorithms. In Table 10 we compare the client resource usage when training a *Large* model with vanilla training and the two SL variants, i.e. plain and U-shaped. Consistent with the findings of Wong et. al., we observed that devices equipped with a GPU exhibit significantly lower CPU usage during training [95]. Offloading a part of the computation to the server benefits all the devices when it comes to CPU and GPU utilization, as these two metrics significantly decrease when using SL if compared to vanilla training. This computation offloading also causes a significant decrease in training times on the devices that lack a GPU, e.g., the round time decreases from 68.06s to 27.84s in the case of Latte Panda Delta 3.

The benefits of SL diminish to some extent when using the U-shaped architecture. In this case, clients need to serialize and deserialize more tensors, transmit more data over the network as discussed in Section 6.2, and run a forward and backward step on the last convolutional layer. The combination of all these factors leads to an increase in training time, as well as higher CPU and GPU involvement.

Devices with GPUs experience longer training times in SL compared to standard local training. This delay arises from various factors including serialization, deserialization, and increased CPU-GPU traffic. However,

[14]We did not optimize the memory usage but rather used the default PyTorch settings.

Table 10. Absolute average CPU and GPU utilization and round training time for training a *Large* model with the two SL variants and vanilla training.

device type	Vanilla Training			Plain SL			U-shaped		
	CPU (%)	GPU (%)	round time (s)	CPU (%)	GPU (%)	round time (s)	CPU (%)	GPU (%)	round time (s)
Jetson AGX Orin	81.05	38.90	3.77	20.37	5.02	22.93	21.15	5.47	27.37
Jetson Orin Nano	80.73	44.29	4.11	24.33	5.86	17.45	25.92	6.25	20.10
Latte Panda Delta3	382.82	0.00	68.06	200.22	0.00	27.84	216.96	0.00	29.68
Orange Pi5B	461.35	0.00	56.39	213.53	0.00	30.39	212.19	0.00	34.47

the primary bottleneck occurs at the central server, which possesses only one GPU and handles simultaneous requests from 14 clients, leading to a significant slowdown in training.

7 Discussion

7.1 Limitations

The presented study compared a diverse set of algorithms by focusing on two key practical aspects of DCML deployments, i.e. model performance and system resource usage. We limited ourselves to these two broad dimensions as DCML is an extremely complex topic and capturing all situations that may arise in practice is close to impossible. For instance, other aspects we did not consider in this paper include:

- *Privacy*: Depending on the type of exchanged data, a "curious" server may discover more or less of the underlying clients' datasets. In the paper, we only reported the original authors' privacy considerations when available. Also, we did not consider differential privacy, so future research could compare algorithms based on their resilience to such artificially-injected noise.
- *Scalability*: Researchers have observed diminishing returns when adding clients beyond a certain point in the FedAvg algorithm [60, 102]. While this observation also applies to the SplitFed $v1$ algorithm, future research should explore whether this property holds for the other considered algorithms.
- *Type of data non-IIDness*: In Section 5.4 we mimicked data non-IIDness by partitioning the datasets according to the Dirichlet distribution, effectively achieving the label-skew type of non-IIDness [69]. However, in real-world scenarios, client datasets may present different types of non-IIDness such as concept shift.
- *Client availability patterns*: Throughout the experimental section we assumed that all clients are available all the time, while in practice, client availability may vary throughout the day [9].
- *Software Heterogeneity*: While the devices in our testbed had various hardware resources, they were homogeneous in terms of the operating system, with all clients running Linux. In other deployments, clients might differ in their underlying software, such as training models on Android and iOS smartphones.
- *Unreliable networks and Client Failures*: The devices in our testbed were connected via a fast (1Gbps) and reliable Ethernet network. In extreme cases, network issues could prevent clients from uploading data in time, potentially leading the server to register client training as a failure. This is not the only cause of client failures.
- *Malicious clients*: Clients may intentionally attempt to compromise the convergence of the algorithm.

We also acknowledge that though we aimed to provide a comparison that is as unbiased and fair as possible, practical DCML deployments are bound to differ to some extent from our setup:

- We conducted the whole experimental section following the most common practice in FL research of training a model from scratch. However, thanks to the abundance of pre-trained models available in many domains, FL fine-tuning of a pre-trained model is also possible [61]. In such a case, several issues discussed throughout the paper get mitigated. For instance, when fine-tuning a model, it is common practice to freeze the initial layers of a model. In such a case, in model-sharing algorithms, clients only need to share the updates of the trainable parameters. It follows, that the amount of exchanged data reduces.
- Even though we attempted to optimize as much as possible the implemented algorithms according to the information at our disposal, we do not claim that the algorithms could not be further improved. For instance, it is possible that models of different sizes would benefit from having different training parameters (e.g., smaller models having larger learning rates and smaller regularization strength). However, this is a dimension we did not consider in this survey as such a case was not explicitly discussed in the original papers where the algorithms were introduced.

One final limitation of our work is that the algorithm comparisons are purely empirical. While we provided intuitive explanations for observed accuracy differences whenever possible, future research could offer a more analytical comparison of the algorithms.

7.2 Impact and Future Work

The experiments conducted in our study revealed that there is no single best algorithm; instead, each algorithm balances computational, network, and accuracy requirements differently. As developing a comprehensive set of use cases to identify the most suitable algorithm for each is challenging, we believe the takeaways listed in this paper provide valuable insights that can assist practitioners in selecting the most appropriate DCML algorithm for their application.

On a high level, we observed that for FL algorithms, sharing model parameters as in the FedAvg, cFedAvg, and HeteroFL algorithms remains the most effective approach for achieving optimal model accuracy. While reducing the shared data volume and fully customizing the model architecture as in the FD, FedKD, and FedMD algorithms may seem appealing, our findings indicate that this often leads to a notable decrease in model accuracy. Additionally, our results demonstrate that the cFedAvg algorithm, despite its simplicity, delivers comparable performance to algorithms explicitly tailored for model-heterogeneous scenarios. This suggests opportunities for advancing and refining model customization algorithms in future research.

However, these results prompt us to consider the extent to which model customization is the solution to the device heterogeneity challenges outlined in this study. Specifically, as repeatedly shown in Section 5, achieving the best accuracy often necessitates large models. Consequently, when employing FL, low-end devices are excluded from this process, as training large models on such devices leads to extended training times and places a significant burden on clients, as demonstrated in Section 6. Conversely, these devices may train a large model when assisted by a server as is the case in SL approaches.

Therefore, given the limitations of both FL and SL algorithms, we believe that to solve the device heterogeneity issues, a promising direction is to view these technologies not as mutually exclusive but rather as complementary to each other. Specifically, while plain SL may not be suitable for large-scale deployment and training large models with FL approaches on low-end devices may be infeasible, integrating SL and FL can harness the strengths of both approaches. For instance, considering the close relationship between the SplitFed and the FedAvg algorithms, one can assign a different number of layers to each device depending on the device's computational availability [74, 97] and hence let every client collaborate in training a large model while minimizing the burden posed on the central server.

8 Conclusion

In this paper, we have explored three distinct families of algorithms designed to facilitate DCML on devices with constrained computing and memory resources. Our investigation revealed that each algorithmic family possesses unique strengths and weaknesses. For instance, FKD algorithms offer model customization by exchanging logits instead of model parameters. However, they often necessitate stateful clients and entail training models from scratch, limiting their competitiveness. SL-based approaches incur high network traffic and computational overhead on the server, while PT-based algorithms enable clients to train models with a size that is proportional to their capacities but may still lag in accuracy compared to the introduced cFedAvg baseline.

To evaluate these algorithms, we conducted experiments in both simulated and live real-world testbeds comprising heterogeneous devices. Our findings underscore the inherent trade-offs between network utilization, model accuracy, and client resource consumption. These results shed light on the complexities involved in optimizing DCML algorithms for diverse device environments and highlight the need for further research to develop more robust and efficient solutions in this domain.

Acknowledgments

This publication is based on work supported by the King Abdullah University of Science and Technology (KAUST) Office of Research Administration (ORA) under Award No. ORA-CRG2021-4699, and by the Slovenian Research Agency through the projects "Context-Aware On-Device Approximate Computing" (J2-3047) and core funding No. P2-0098. For computer time, this research used the resources of the Supercomputing Laboratory at KAUST. We are thankful to Amândio R. Faustino for his support in conducting the experiments.

References

[1] Ahmed M. Abdelmoniem, Chen-Yu Ho, Pantelis Papageorgiou, and Marco Canini. 2022. Empirical Analysis of Federated Learning in Heterogeneous Environments. In *EuroMLSys*.

[2] Ahmed M. Abdelmoniem, Atal Narayan Sahu, Marco Canini, and Suhaib A. Fahmy. 2023. REFL: Resource-Efficient Federated Learning. In *EuroSys*.

[3] Sharif Abuadbba, Kyuyeon Kim, Minki Kim, Chandra Thapa, Seyit Ahmet Çamtepe, Yansong Gao, Hyoungshick Kim, and Surya Nepal. 2020. Can We Use Split Learning on 1D CNN Models for Privacy Preserving Training?. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*.

[4] Samiul Alam, Luyang Liu, Ming Yan, and Mi Zhang. 2022. FedRolex: Model-Heterogeneous Federated Learning with Rolling Sub-Model Extraction. In *NeurIPS*.

[5] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Róbert Ormándi, George E. Dahl, and Geoffrey E. Hinton. 2018. Large Scale Distributed Neural Network Training Through Online Distillation. In *ICLR*.

[6] Manoj Ghuhan Arivazhagan, Vinay Aggarwal, Aaditya Kumar Singh, and Sunav Choudhary. 2019. Federated Learning with Personalization Layers. (2019). arXiv:1912.00818 [cs.DC]

[7] Gustav A. Baumgart, Jaemin Shin, Ali Payani, Myungjin Lee, and Ramana Rao Kompella. 2024. Not All Federated Learning Algorithms Are Created Equal: A Performance Evaluation Study. (2024). arXiv:2403.17287 [cs.DC]

[8] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. 2020. Flower: A Friendly Federated Learning Research Framework. (2020). arXiv:2007.14390 [cs.DC]

[9] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards Federated Learning at scale: System design. (2019). arXiv:1902.01046 [cs.DC]

[10] Janez Božič, Amândio R. Faustino, Boris Radovič, Marco Canini, and Veljko Pejović. 2024. Where is the Testbed for my Federated Learning Research?. In *ACM/IEEE Symposium on Edge Computing (SEC)*.

[11] Christopher Briggs, Zhong Fan, and Peter Andras. 2020. Federated Learning With Hierarchical Clustering of Local Updates To Improve Training on Non-Iid Data. In *International Joint Conference on Neural Networks (IJCNN)*.

[12] Cristian Buciluă, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model Compression. In *SIGKDD*.

[13] S Caldas, J Konečny, H B McMahan, and others. 2018. Expanding the Reach of Federated Learning by Reducing Client Resource Requirements. (2018). arXiv:1812.07210 [cs.DC]

[14] Hongyan Chang, Virat Shejwalkar, Reza Shokri, and Amir Houmansadr. 2019. Cronus: Robust and Heterogeneous Collaborative Learning with Black-Box Knowledge Transfer. (2019). arXiv:1912.11279 [cs.DC]

[15] Zachary Charles, Kallista A. Bonawitz, Stanislav Chiknavaryan, Brendan McMahan, and Blaise Agüera y Arcas. 2022. Federated Select: A Primitive for Communication- and Memory-Efficient Federated Learning. (2022). arXiv:2208.09432 [cs.DC]

[16] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. 2022. THE-X: Privacy-Preserving Transformer Inference with Homomorphic Encryption. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

[17] Gary Cheng, Zachary Charles, Zachary Garrett, and Keith Rush. 2022. Does Federated Dropout actually work?. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

[18] Sijie Cheng, Jingwen Wu, Yanghua Xiao, and Yang Liu. 2021. FedGEMS: Federated Learning of Larger Server Models via Selective Knowledge Fusion. (2021). arXiv:2110.11027 [cs.DC]

[19] Yae Jee Cho, Andre Manoel, Gauri Joshi, Robert Sim, and Dimitrios Dimitriadis. 2022. Heterogeneous Ensemble Knowledge Transfer for Training Large Models in Federated Learning. In *International Joint Conferences on Artificial Intelligence (IJCAI)*.

[20] Yae Jee Cho, Jianyu Wang, Tarun Chirvolu, and Gauri Joshi. 2023. Communication-Efficient and Model-Heterogeneous Personalized Federated Learning via Clustered Knowledge Transfer. *IEEE Journal of Selected Topics in Signal Processing* (2023).

[21] Ayush Chopra, Surya Kant Sahu, Abhishek Singh, Abhinav Java, Praneeth Vepakomma, Vivek Sharma, and Ramesh Raskar. 2021. AdaSplit: Adaptive Trade-offs for Resource-constrained Distributed Deep Learning. arXiv:2112.01637 [cs.LG]

[22] Luke Nicholas Darlow, Elliot J. Crowley, Antreas Antoniou, and Amos J. Storkey. 2018. CINIC-10 Is Not ImageNet or CIFAR-10. (2018). arXiv:1810.03505 [cs.DC]

[23] Enmao Diao, Jie Ding, and Vahid Tarokh. 2021. HeteroFL: Computation and Communication Efficient Federated Learning for Heterogeneous Clients. In *ICLR*.

[24] Qiang Duan, Shijing Hu, Ruijun Deng, and Zhihui Lu. 2022. Combined Federated and Split Learning in Edge Computing for Ubiquitous Intelligence in Internet of Things: State-of-the-Art and Future Directions. *Sensors* (2022).

[25] Chen Dun, Mirian Hipolito Garcia, Chris Jermaine, Dimitrios Dimitriadis, and Anastasios Kyrillidis. 2023. Efficient and Light-Weight Federated Learning via Asynchronous Distributed Dropout. In *AISTATS*.

[26] Chen Dun, Cameron R. Wolfe, Christopher M. Jermaine, and Anastasios Kyrillidis. 2022. ResIST: Layer-wise decomposition of ResNets for distributed training. In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

[27] Wenzhi Fang, Dong-Jun Han, and Christopher G. Brinton. 2024. Submodel Partitioning in Hierarchical Federated Learning: Algorithm Design and Convergence Analysis. In *IEEE International Conference on Communication (ICC)*.

[28] Yansong Gao, Minki Kim, Sharif Abuadbba, Yeonjae Kim, Chandra Thapa, Kyuyeon Kim, Seyit Ahmet Çamtepe, Hyoungshick Kim, and Surya Nepal. 2020. End-to-End Evaluation of Federated Learning and Split Learning for Internet of Things. In *International Symposium on Reliable Distributed Systems (SRDS)*.

[29] Yansong Gao, Minki Kim, Chandra Thapa, Alsharif Abuadbba, Zhi Zhang, Seyit Camtepe, Hyoungshick Kim, and Surya Nepal. 2022. Evaluation and Optimization of Distributed Machine Learning Techniques for Internet of Things. *IEEE Trans. Comput.* (2022).

[30] Manish Gawali, C. S. Arvind, Shriya Suryavanshi, Harshit Madaan, Ashrika Gaikwad, K. N. Bhanu Prakash, Viraj Kulkarni, and Aniruddha Pant. 2021. Comparison of Privacy-Preserving Distributed Deep Learning Methods in Healthcare. In *Medical Image Understanding and Analysis (MIUA)*.

[31] Avishek Ghosh, Jichan Chung, Dong Yin, and Kannan Ramchandran. 2020. An Efficient Framework for Clustered Federated Learning. In *NeurIPS*.

[32] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*.

[33] Google. 2022. How Messages Improves Suggestions With Federated Technology. https://support.google.com/messages/answer/9327902?hl=en. Accessed: 2023-11-11.

[34] Otkrist Gupta and Ramesh Raskar. 2018. Distributed Learning of Deep Neural Network Over Multiple Agents. *Journal of Network and Computer Applications* (2018).

[35] Andrew Hard, Kanishka Rao, Rajiv Mathews, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. Federated Learning for Mobile Keyboard Prediction. (2018). arXiv:1811.03604

[36] Chaoyang He, Murali Annavaram, and Salman Avestimehr. 2020. Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge. In *NeurIPS*.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[38] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. (2015). arXiv:1503.02531 [cs.DC]

[39] Samuel Horváth, Stefanos Laskaridis, Mário Almeida, Ilias Leontiadis, Stylianos I. Venieris, and Nicholas D. Lane. 2021. FjORD: Fair and Accurate Federated Learning under heterogeneous targets with Ordered Dropout. In *NeurIPS*.

[40] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip B. Gibbons. 2020. The Non-IID Data Quagmire of Decentralized Machine Learning. In *ICML*.

[41] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. 2019. Measuring the Effects of Non-Identical Data Distribution for Federated Visual Classification. (2019). arXiv:1909.06335 [cs.DC]

[42] Li Hu, Hongyang Yan, Lang Li, Zijie Pan, Xiaozhang Liu, and Zulong Zhang. 2021. MHAT: an Efficient Model-Heterogenous Aggregation Training Scheme For Federated Learning. *Information Sciences* (2021).

[43] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[44] Sohei Itahara, Takayuki Nishio, Yusuke Koda, Masahiro Morikura, and Koji Yamamoto. 2023. Distillation-Based Semi-Supervised Federated Learning for Communication-Efficient Collaborative Training With Non-IID Private Data. *IEEE Transactions on Mobile Computing* (2023).

[45] Eunjeong Jeong, Seungeun Oh, Hyesung Kim, Jihong Park, Mehdi Bennis, and Seong-Lyun Kim. 2018. Communication-Efficient On-Device Machine Learning: Federated Distillation and Augmentation under Non-IID Private Data. (2018). arXiv:1811.11479

[46] Yuang Jiang, Shiqiang Wang, Víctor Valls, Bong Jun Ko, Wei-Han Lee, Kin K. Leung, and Leandros Tassiulas. 2023. Model Pruning Enables Efficient Federated Learning on Edge Devices. *IEEE Transactions on Neural Networks and Learning Systems* (2023).

[47] Zhida Jiang, Yang Xu, Hongli Xu, Zhiyuan Wang, Jianchun Liu, Chen Qian, and Chunming Qiao. 2024. Computation and Communication Efficient Federated Learning With Adaptive Model Pruning. *IEEE Transactions on Mobile Computing* (2024).

[48] Minsu Kim, Alexander C. DeRieux, and Walid Saad. 2023. A Bargaining Game for Personalized, Energy Efficient Split Learning over Wireless Networks. In *IEEE Wireless Communications and Networking Conference (WCNC)*.

[49] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, et al. 2016. Overcoming Catastrophic Forgetting in Neural Networks. (2016). arXiv:1612.00796

[50] A Krizhevsky. 2009. Learning Multiple Layers of Features From Tiny Images. (2009).

[51] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. 2021. Hermes: An Efficient Federated Learning Framework for Heterogeneous Mobile Clients. In *MobiCom*.

[52] Daliang Li and Junpu Wang. 2019. FedMD: Heterogenous Federated Learning via Model Distillation. (2019). arXiv:1910.03581 [cs.DC]

[53] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2020. Federated Optimization in Heterogeneous Networks. In *MLSys*.

[54] Paul Pu Liang, Terrance Liu, Ziyin Liu, Ruslan Salakhutdinov, and Louis-Philippe Morency. 2020. Think Locally, Act Globally: Federated Learning with Local and Global Representations. (2020). arXiv:2001.01523 [cs.DC]

[55] Yunming Liao, Yang Xu, Hongli Xu, Zhiwei Yao, Lun Wang, and Chunming Qiao. 2023. Accelerating Federated Learning With Data and Model Parallelism in Edge Computing. *IEEE/ACM Transactions on Networking* (2023).

[56] Tao Lin, Lingjing Kong, Sebastian U. Stich, and Martin Jaggi. 2020. Ensemble Distillation for Robust Model Fusion in Federated Learning. In *NeurIPS*.

[57] Terrance Liu and Paul Liang. 2020. Federated Learning with Local and Global Representations. https://github.com/pliang279/LG-FedAvg. Accessed: 2024-03-06.

[58] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS*.

[59] Muhammad Tahir Munir, Muhammad Mustansar Saeed, Mahad Ali, Zafar Ayyub Qazi, Agha Ali Raza, and Ihsan Ayyub Qazi. 2023. Learning Fast and Slow: Towards Inclusive Federated Learning. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*.

[60] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. 2022. Federated Learning with Buffered Asynchronous Aggregation. In *AISTATS*.

[61] John Nguyen, Jianyu Wang, Kshitiz Malik, Maziar Sanjabi, and Michael G. Rabbat. 2023. Where to Begin? On the Impact of Pre-Training and Initialization in Federated Learning. In *ICLR*.

[62] Yue Niu, Saurav Prakash, Souvik Kundu, Sunwoo Lee, and Salman Avestimehr. 2022. Federated Learning of Large Models at the Edge via Principal Sub-Model Training. (2022). arXiv:2208.13141 [cs.DC]

[63] Ziru Niu, Hai Dong, and A. Kai Qin. 2024. FedSPU: Personalized Federated Learning for Resource-constrained Devices with Stochastic Parameter Update. (2024). arXiv:2403.11464 [cs.DC]

[64] Seungeun Oh, Jihong Park, Praneeth Vepakomma, Sihun Baek, Ramesh Raskar, Mehdi Bennis, and Seong-Lyun Kim. 2022. LocFedMix-SL: Localize, Federate, and Mix for Improved Scalability, Convergence, and Latency in Split Learning. In *WWW*.

[65] Shraman Pal, Mansi Uniyal, Jihong Park, Praneeth Vepakomma, Ramesh Raskar, Mehdi Bennis, Moongu Jeon, and Jinho Choi. 2021. Server-Side Local Gradient Averaging and Learning Rate Acceleration for Scalable Split Learning. (2021). arXiv:2112.05929 [cs.DC]

[66] Kamalesh Palanisamy, Vivek Khimani, Moin Hussain Moti, and Dimitris Chatzopoulos. 2021. SplitEasy: A Practical Approach for Training ML models on Mobile Devices. In *HotMobile*.

[67] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-Aware Quantization for Training and Inference of Neural Networks. In *European Conference on Computer Vision (ECCV)*.

[68] Dario Pasquini, Giuseppe Ateniese, and Massimo Bernaschi. 2021. Unleashing the Tiger: Inference Attacks on Split Learning. In *ACM Special Interest Group on Security, Audit and Control (SIGSAC)*.

[69] Peter Kairouz et.al. 2021. Advances and Open Problems in Federated Learning. *Foundations and Trends® in Machine Learning* (2021).

[70] Krishna Pillutla, Kshitiz Malik, Abdelrahman Mohamed, Michael G. Rabbat, Maziar Sanjabi, and Lin Xiao. 2022. Federated Learning with Partial Model Personalization. In *ICML*.

[71] Maarten G. Poirot, Praneeth Vepakomma, Ken Chang, Jayashree Kalpathy-Cramer, Rajiv Gupta, and Ramesh Raskar. 2019. Split Learning for Collaborative Deep Learning in Healthcare. (2019). arXiv:1912.12115 [cs.DC]

[72] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. 2021. Adaptive Federated Optimization. In *ICLR*.

[73] Steffen Rendle. 2010. Factorization Machines. In *International Conference on Data Mining (ICDM)*.

[74] Eric Samikwa, Antonio Di Maio, and Torsten Braun. 2022. ARES: Adaptive Resource-Aware Split Learning for Internet of Things. *Comput. Networks* (2022).

[75] Felix Sattler, Tim Korjakow, Roman Rischke, and Wojciech Samek. 2023. FedAUX: Leveraging Unlabeled Auxiliary Data in Federated Learning. *IEEE Transactions on Neural Networks and Learning Systems* (2023).

[76] Felix Sattler, Klaus-Robert Müller, and Wojciech Samek. 2021. Clustered Federated Learning: Model-Agnostic Distributed Multitask Optimization Under Privacy Constraints. *IEEE Transactions on Neural Networks and Learning Systems* (2021).

[77] Tao Shen, Jie Zhang, Xinkang Jia, Fengda Zhang, Gang Huang, Pan Zhou, Kun Kuang, Fei Wu, and Chao Wu. 2020. Federated mutual learning. (2020). arXiv:2006.16765 [cs.LG]

[78] Nir Shlezinger, Mingzhe Chen, Yonina C. Eldar, H. Vincent Poor, and Shuguang Cui. 2021. UVeQFed: Universal Vector Quantization for Federated Learning. *IEEE Transactions on Signal Processing* (2021).

[79] Egor Shulgin and Peter Richtárik. 2024. Towards a Better Theoretical Understanding of Independent Subnetwork Training. In *ICML*.

[80] Dan Simmons. 2022. 17 Countries with GDPR-like Data Privacy Laws. https://insights.comforte.com/countries-with-gdpr-like-data-privacy-laws. Accessed: 2023-12-06.

[81] Abhishek Singh, Praneeth Vepakomma, Otkrist Gupta, and Ramesh Raskar. 2019. Detailed Comparison of Communication Efficiency of Split Learning And Federated Learning. (2019). arXiv:1909.09145

[82] Karan Singhal, Hakim Sidahmed, Zachary Garrett, Shanshan Wu, John Rush, and Sushant Prakash. 2021. Federated Reconstruction: Partially Local Federated Learning. In *NeurIPS*.

[83] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. 2015. Striving for Simplicity: The All Convolutional Net. In *ICLR*.

[84] Sebastian U Stich. 2018. Local SGD Converges Fast and Communicates Little. (2018). arXiv:1805.09767 [cs.DC]

[85] Alysa Ziying Tan, Han Yu, Lizhen Cui, and Qiang Yang. 2023. Towards Personalized Federated Learning. *IEEE Transactions on Neural Networks and Learning Systems* (2023).

[86] Yue Tan, Guodong Long, Lu Liu, Tianyi Zhou, Qinghua Lu, Jing Jiang, and Chengqi Zhang. 2022. FedProto: Federated Prototype Learning across Heterogeneous Clients. In *AAAI*.

[87] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, Seyit Camtepe, and Lichao Sun. 2022. SplitFed: When Federated Learning Meets Split Learning. In *AAAI*.

[88] Nguyen Truong, Kai Sun, Siyao Wang, Florian Guitton, and Yike Guo. 2021. Privacy Preservation in Federated Learning: An Insightful Survey From The GDPR Perspective. *Computers & Security* (2021).

[89] Valeria Turina, Zongshun Zhang, Flavio Esposito, and Ibrahim Matta. 2020. Combining Split and Federated Architectures for Efficiency and Privacy In Deep Learning. In *CoNEXT*.

[90] Saeed Vahidian, Mahdi Morafah, and Bill Lin. 2021. Personalized Federated Learning by Structured and Unstructured Pruning under Data Heterogeneity. In *IEEE International Conference on Distributed Computing Systems (ICDCS) Workshops*.

[91] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2018. Split Learning for Health: Distributed Deep Learning Without Sharing Raw Patient Data. (2018). arXiv:1812.00564 [cs.DC]

[92] Dingzhu Wen, Ki Jun Jeon, and Kaibin Huang. 2022. Federated Dropout - A Simple Approach for Enabling Federated Learning on Resource Constrained Devices. *IEEE Wireless Communications Letters* (2022).

[93] Herbert Woisetschläger, Alexander Isenko, Ruben Mayer, and Hans-Arno Jacobsen. 2023. FLEDGE: Benchmarking Federated Machine Learning Applications in Edge Computing Systems. (2023). arXiv:2306.05172

[94] Cameron R. Wolfe, Jingkang Yang, Fangshuo Liao, Arindam Chowdhury, Chen Dun, Artun Bayer, Santiago Segarra, and Anastasios Kyrillidis. 2024. GIST: Distributed Training for Large-Scale Graph Convolutional Networks. *Journal of Applied and Computational Topology* (2024).

[95] Kok-Seng Wong, Manh Nguyen-Duc, Khiem Le-Huy, et al. 2023. An Empirical Study of Federated Learning on IoT-Edge Devices: Resource Allocation and Heterogeneity. (2023). arXiv:2305.19831

[96] Chuhan Wu, Fangzhao Wu, Lingjuan Lyu, Yongfeng Huang, and Xing Xie. 2022. Communication-Efficient Federated Learning via Knowledge Distillation. *Nature Communications* (2022).

[97] Di Wu, Rehmat Ullah, Paul Harvey, Peter Kilpatrick, Ivor T. A. Spence, and Blesson Varghese. 2022. FedAdapt: Adaptive Offloading for IoT Devices in Federated Learning. *IEEE Internet of Things Journal* (2022).
[98] Chenhao Xu, Youyang Qu, Yong Xiang, and Longxiang Gao. 2023. Asynchronous Federated Learning on Heterogeneous Devices: A Survey. *Computer Science Review* (2023).
[99] Wenyuan Xu, Weiwei Fang, Yi Ding, Meixia Zou, and Naixue Xiong. 2021. Accelerating Federated Learning for IoT in Big Data Analytics With Pruning, Quantization and Selective Updating. *IEEE Access* (2021).
[100] Mark Xue and Julien Freudiger. 2019. Designing for Privacy. https://developer.apple.com/videos/play/wwdc2019/708. Accessed: 2023-11-11.
[101] Liping Yi, Gang Wang, Xiaoguang Liu, Zhuan Shi, and Han Yu. 2023. FedGH: Heterogeneous federated learning with generalized global header. In *ACM International Conference on Multimedia.*
[102] Ashkan Yousefpour, Shen Guo, Ashish Shenoy, Sayan Ghosh, Pierre Stock, Kiwan Maeng, Schalk-Willem Krüger, Michael G. Rabbat, Carole-Jean Wu, and Ilya Mironov. 2023. Green Federated Learning. (2023). arXiv:2303.14604 [cs.DC]
[103] Binhang Yuan, Cameron R. Wolfe, Chen Dun, Yuxin Tang, Anastasios Kyrillidis, and Chris Jermaine. 2022. Distributed Learning of Fully Connected Neural Networks using Independent Subnet Training. *Proceedings of the VLDB Endowment* (2022).
[104] Mikhail Yurochkin, Mayank Agarwal, Soumya Ghosh, Kristjan Greenewald, Nghia Hoang, and Yasaman Khazaeni. 2019. Bayesian Nonparametric Federated Learning of Neural Networks. In *ICML*.
[105] Jie Zhang, Song Guo, Xiaosong Ma, Haozhao Wang, Wenchao Xu, and Feijie Wu. 2021. Parameterized Knowledge Transfer for Personalized Federated Learning. In *NeurIPS*.
[106] Shiqiang Zhang, Zihang Zhao, Detian Liu, Yang Cao, Hengliang Tang, and Siqing You. 2025. Edge-assisted U-shaped split federated learning with privacy-preserving for Internet of Things. *Expert Systems with Applications* (2025).
[107] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level Convolutional Networks for Text Classification. In *NeurIPS*.
[108] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. 2018. Federated Learning with Non-IID Data. (2018). arXiv:1806.00582 [cs.DC]

A Experimental setup

In this section we detail the experimental setup used throughout the evaluation phase of the paper.
Software stack: We implemented all the FL algorithms using the Flower FL framework [8], with Pytorch as the deep learning library. In order to be able to test the SL algorithms, we also developed an extension to Flower, which is available on https://github.com/sands-lab/slower. For managing and automating the deployment of FL algorithms to the physical devices, we developed a custom library, which uses Kubernetes for deploying clients as containers on the devices.
Data partitioning: Throughout the paper we used the CIFAR10, CIFAR100, and CINIC10 dataset. For the CIFAR10 and CIFAR100 we use the 60000 images composing both the training and test set, i.e., we do not differentiate between the train and test partition in the original dataset. Conversely, in the CINIC10 dataset, we use the 90000 images in the train dataset. When constructing the client datasets, we first divide the original dataset into N partitions. Then, we reserve 15% of the so-obtained dataset partitions for validation and 15% for the test dataset.

For algorithms that require an additional public dataset, we use a random sample of 1000 images from the CIFAR100 dataset as a public dataset.

The experiments in Section 5.3, Section 5.4, and Section 5.2 involved 21 clients grouped into three tiers: low-capacity, medium-capacity, and high-capacity, with seven clients in each tier. In contrast, the experiments in Section 5.5 were conducted with 20 clients.
Models: We use a slightly adapted version of the fully convolutional model referred to as “Model C” in [83], which has been applied also in FL experiments, e.g. [13]. The exact model architecture is stated in Table 11. We employed three different model architectures derived from the base model (*Large*) by reducing the number of channels in convolutional layers. Each client was assigned to one of the three available classes of devices, i.e. low-, medium-, and high-end, and depending on this membership, trained a *Small* (52823 parameters), *Medium* (313786 parameters), or *Large* (1249642 parameters) model. Considering that all three models have the very same structure, we can directly use them also for PT approaches.

Table 11. Employed model architectures. All convolutional layers are followed by a ReLU activation.

Layer	Kernel size	Stride	Padding	Output channels		
				Small	Medium	Large
Conv2D	3×3	1	1	20	48	96
Conv2D	3×3	1	1	20	48	96
MaxPool2D	3×3	2				
Conv2D	3×3	1	1	39	96	192
Conv2D	3×3	1	1	39	96	192
MaxPool2D	3×3	2	1			
Conv2D	3×3	1	1	39	96	192
Conv2D	3×3	1	1	39	96	192
Conv2D	1×1	1	1	10	10	10
AvgPool2D	6×6	1	0			

Table 12. Hyperparameters search space. We **bold** the values that yield the highest accuracy for two local training epochs. All the algorithms also used weight decay 3e-4 and gradient norm clipping 4.0. B is the batch size, η the learning rate, and μ the FedProx [53] regularization strength.

Algorithm	Tuned hyperparameters	Values	Constant hyperparameters
FedAvg [58]	η B	[0.02, **0.05**, 0.1] [8, **12**, 16]	/
FD [45]	Temperature KD strength	[0.5, **1.0**, 2.0, 4.0] [0.1, 0.2, **0.5**]	B=8, η=0.05
FedMD [52]	Public dataset size Temperature	[1000, **4000**] [0.5, **1.0**, 2.0]	B=12, η=0.05
FedDF [56]	Server training epochs Weight predictions	[**1**, 2] [**true**, false]	B=8, η=0.05 μ=0.001
Federated Dropout [13]	η B	[0.05, **0.1**] [**8**, 12, 16]	/
HeteroFL [23]	η B	[0.05, **0.1**] [**8**, 12, 16]	/
LgFedAvg [54]	η B	[**0.05**, 0.1] [8, **12**, 16]	/

Hyperparameter tuning: for all the algorithms we performed a grid search over possible values of hyper-parameters to determine the configuration, that achieves the highest accuracy. We spent approximately the same amount of time in hyperparameter tuning for all the algorithms to provide a fair comparison (≈1 day for each method using 42 CPUs and 2 GPUs). We report in Table 12 the exact hyperparameters we tested for every algorithm. For optimization, in all the algorithms we used the plain SGD optimizer with no momentum. We used SGD as it does not incur additional memory usage by the client such as optimizer state.

A.1 Additional comments about algorithm implementation

- **FD**: We were not able to reproduce the results stated in [45]. We contacted the authors to ask for clarification but received no answer. Also, the algorithm stated in the paper does not consider cases in which some clients are missing certain target labels and cases in which a client fails to upload its logits to the server. We avoid both these issues by computing the global logits for a class as the average logits across all clients, in place of the average across all clients excluding the target client. In the performed experiments, this update does not degrade the performance of the model.
- **FedKD**: We ignore the adaptive hidden loss (Equation 5. in [96]), as it primarily targets transformer-like architectures. Regarding the architecture of the globally shared model, we used a model with 10% of the original filters (that is, the convolutional layers in Table 11 have either 9 or 19 filters). While the results could be improved by using a larger model, this would conflict with the constraints of the low-capacity clients. We did not implement the Dynamic Gradient Approximation extension as it is orthogonal to the training task.
- **LgFedAvg**: We assume that the model can be divided into an encoder, which we set to comprise the first five convolutional layers, and a classification head, comprising the remaining two convolutional layers. To be able to run the model across all clients, we set the parameters of the final two layers to be equal to the ones of the *Small* model. The first four layers of the encoder are equal to the architectures stated in Table 11, while the last encoder layer has in all cases 39 output filters because its output needs to be passed to the classification head.
- **FedMD**: we did not include the pre-training stage performed on the public dataset. This step is orthogonal to the remainder of the algorithm and might easily be integrated into other algorithms as well. During the KD stage, we used the KL loss with temperature 1.0.
- **SplitFed**: We divide the model into a client-side (first two convolutional layers) and a server-side segment, comprising the remaining five layers. We train a *Large* model as we assume that the server has no computational constraints. Also, we reduce the number of filters in the first convolutional layer to 64, so that the first two layers have approximately the same number of parameters of the *Small* model. We implemented the *SFLV1* algorithm as in the original paper [87] and an extension that uses the U-shaped architecture. In this case, the clients contain, apart from the initial layers as in the plain version, also the uppermost layer, and these layers are averaged across all devices after every server training round.
- **Partial training**: For PT approaches (HeteroFL and Federated Dropout) we found that gradient clipping has a vital role, as without it small models (e.g. model obtained by dropping 80% of the channels) easily diverge during local training. This confirms the statement made in [23], wherein authors state that “gradient clipping stabilizes the optimization”. Therefore, in partial training approaches we decrease the gradient norm clipping parameter to 1.0 whenever the model dropout rate is lower than 1.0.
- **Server-side data aggregation**: For algorithms that support heterogeneous models and involve some form of averaging on the server (FedMD, FedDF), we tried weighting the data sent by the client with the weights 0.5 (*Large* model), 0.35 (*Medium* mode), and 0.15 (*Small* model). We did not optimize these values, but rather made an educated guess and picked a reasonable choice in which more powerful models are given more weight. This adoption consistently outperforms the non-weighted case, though the difference is typically limited to $0.5 - 1.0\%$.

B Results on other vision datasets

We report the accuracy w.r.t. dataset size for the CIFAR100 dataset in Figure 8 and for the CINIC dataset in Figure 9. The results for the CIFAR100 dataset are in line with the ones discussed in Section 5.3, with SL consistently yielding the highest accuracy and a significant gap between model-sharing algorithms and algorithms, that do

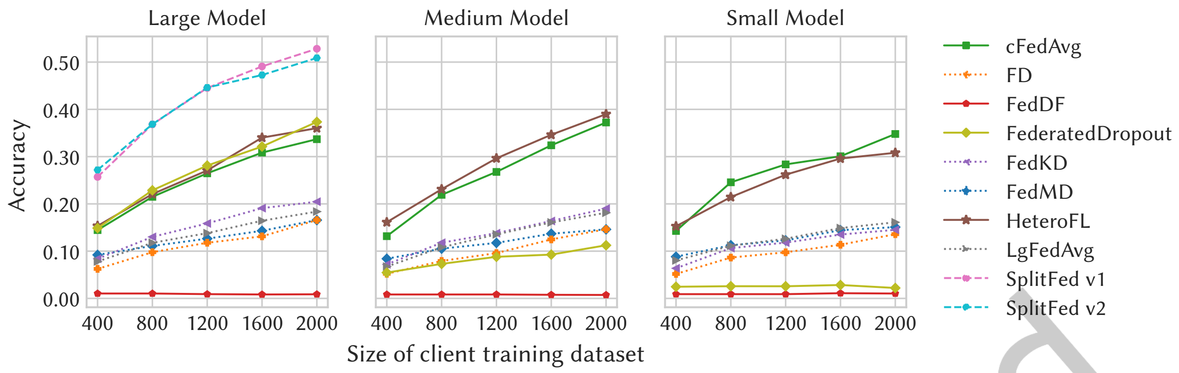

Fig. 8. Average client test accuracy w.r.t. the training dataset size and model size on the CIFAR100 dataset. The experiment is equivalent to the one described in Figure 2.

not share model parameters. The FedDF algorithm never converges, as two local training epochs are not enough for the training to produce good enough models. Note also, that the cFedAvg algorithm consistently outperforms the HetefoFL algorithm for the *Small* model case, confirming the fact, that PT algorithms have difficulties training very small models as observed in Section 5.5.

In contrast to the results obtained on the CIFAR10 and CIFAR100 datasets, in the case of the CINIC dataset there no longer is a clear difference between stateful and stateless clients. However, the individual trends for the algorithms are consistent with the ones discussed in Section 5.2:

- **HeteroFL**: the algorithm consistently yields the best accuracy for the *Large* and *Medium* models, however, the efficiency of the algorithm decreases for large P in the low-capacity cluster training the *Small* model;
- **cFedAvg**: the baseline provides relatively high accuracy in all data settings, as it is always among the three algorithms with the highest accuracy;
- **FedKD**: the algorithm yields unsatisfactory results when P is small, however, the performance of the algorithm quickly improves as P increases. In the end, the accuracy of the FedKD algorithm in the low-capacity cluster is the highest for any $P \geq 1500$, while in the mid- and high-capacity cluster, the algorithm is among the three algorithms with the highest accuracy for $P \geq 2000$.
- **FedDF**: the algorithm does not converge for $P = 500$;
- **FD**: The accuracy of this algorithm is low for small dataset sizes, however, the performance improves so that in the end, in the low-capacity cluster it is the third best option for $P = 2500$;
- **FedMD**: the accuracy of this algorithm is the best among the stateful clients for $P = 500$.

The reduced discrepancy in accuracy between stateful and stateless algorithms can be attributed to the unique properties of the CINIC dataset. CINIC extends the CIFAR10 dataset by including images from ImageNet, resulting in a noisier dataset with a mix of easily classifiable and challenging images. This leads to a lesser impact of additional data on accuracy compared to other datasets. The observation is confirmed in Table 13, where we see that training models on individual client datasets without collaboration yields similar test accuracy for CIFAR10 and CINIC. However, when centralizing the 20 datasets and hence centrally training a model on a dataset with size $20 \cdot 1500 = 30000$, the accuracy increases by only 12.36% for the CINIC dataset, while the accuracy improvement is more than double (26.82%) in the case of the CIFAR10 dataset. Note, that these experiments were conducted using a traditional centralized learning approach.

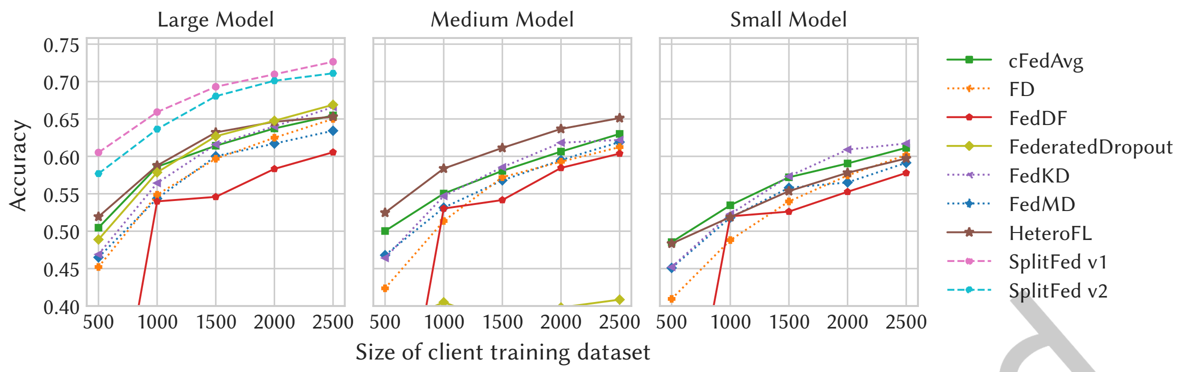

Fig. 9. Average client test accuracy w.r.t. the training dataset size and model size on the CINIC dataset. The experiment is equivalent to the one described in Figure 2.

Table 13. Test accuracy (in %) obtained by training a model on each of the 20 client datasets with 1500 data points each and on the combined dataset with 30000 data points.

	CIFAR10	CIFAR100	CINIC10
Private training by clients	54.29 ± 0.02	9.94 ± 0.01	55.35 ± 0.04
Centralized training with aggregated dataset	81.11 ± 0.01	46.34 ± 0.01	67.71 ± 0.01

Table 14. Summary of models used for text classification.

	Embedding dimension	Intermediate size	Parameter count
Small model	32	48	126724
Medium model	64	96	341508
Large model	160	256	1545060

C Results on text classification task

We here present the accuracy achieved by the considered algorithms on a text classification task. In particular, we use the Ag-News dataset [107], which involves classifying news descriptions into one of the four available topics, i.e., “World”, “Business”, “Sports”, and “Sci/Tech”.

We employ a BERT-like architectures for this task. As shown in Table 14, we construct different model sizes by varying the embedding dimension (also called “hidden size”) and the intermediate size, i.e., the dimensionality of the “intermediate” feed-forward layer in the transformer encoder. All other model configuration is kept constant: the number of attention heads is set to 8, the vocabulary size is set to 2000, and the number of hidden layers is set to 6. For training, we use the Adam optimizer with the learning rate set to $2e-4$ and batch size set to 32 for all the algorithms.

We report in Figure 10 the results with respect to the dataset size. Notably, we observe that the main conclusions made for the image classification tasks transfer to the discussed text-classification setting:

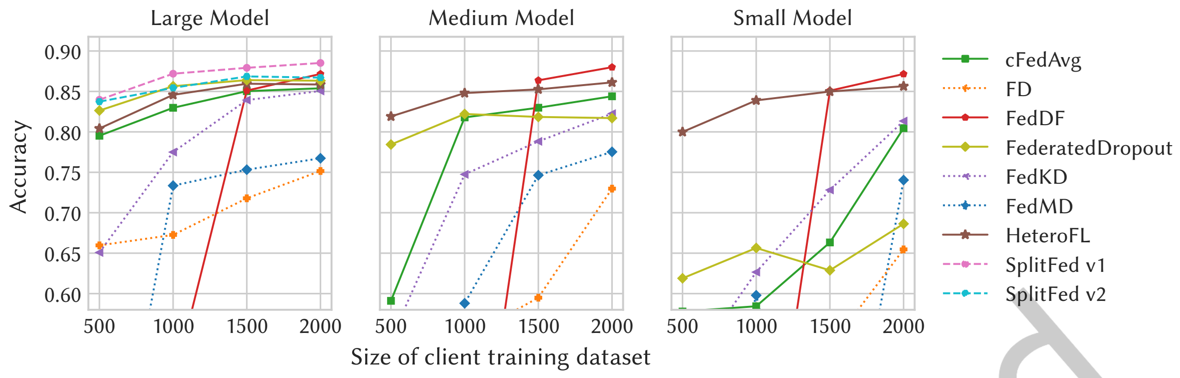

Fig. 10. Average client test accuracy w.r.t. the training dataset size and model size on the Ag-News dataset. The experiment is equivalent to the one described in Figure 2.

- The SplitFed algorithms outperform all other algorithms, with SplitFed v1 demonstrating better performance than SplitFed v2.
- In the PT family of algorithms, HeteroFL achieves significantly higher accuracy than Federated Dropout for all model sizes except the *Large* model. Moreover, Federated Dropout yields particularly low accuracy when using small models.
- The FedDF algorithm fails to converge when the client's training set is too small, as the KD stage on the server becomes detrimental if the client models are undertrained. However, this issue could be mitigated to some extent by increasing the number of local training epochs on the clients.
- Stateful algorithms (FD, FedMD, FedKD) exhibit a significant accuracy drop compared to stateless ones.
- For algorithms that require a dataset on the server (e.g., FedDF, FedMD), we observe a significant accuracy drop when using a dataset with different characteristics from the client datasets (e.g., IMDB and Reuters datasets). The accuracy loss can be as severe as 10%. Therefore, to obtain the accuracy shown in Figure 10, we used a subset of the Ag-News dataset as the public dataset.
- FedKD achieves very high accuracy, especially with larger datasets, and in some cases matches the performance of stateless algorithms. However, in our experiments, the collectively trained model was set to the same size as the *Small* model. This implies that weak clients had to train twice as many parameters compared to the other algorithms. We use such a model as the publicly shared model because we observed that training extremely small BERT models (e.g., with an embedding dimension of 16) is unstable with FL – consider for instance the difference in accuracy between training a *Large* and *Small* model with cFedAvg. Additionally, FedKD required a very high number of training rounds to converge. For example, with dataset sizes of 1500, FedKD took 340 server rounds to converge, whereas HeteroFL converged in just 36 rounds.[15]

The main difference between the text classification and image classification results lies in the poorer performance of the cFedAvg algorithm compared to other stateless algorithms. This difference is particularly pronounced when dataset sizes are small, underscoring the importance of having a large number of clients and a substantial amount of data when training BERT-like models.

Received 22 April 2024; revised 20 September 2024; accepted 30 November 2024

[15]Recall, that we consider an algorithm to have converged if the average validation accuracy does not improve for four consecutive evaluation rounds. For the text classification task, we evaluate the global model every four training rounds.