Journal of Machine Learning Research 25 (2024) 1-39 Submitted 2/23; Revised 4/24; Published 11/24

Federated Automatic Differentiation

Keith Rush KRUSH@GOOGLE.COM
Google Research
Seattle, WA, USA

Zachary Charles ZACHCHARLES@GOOGLE.COM
Google Research
Seattle, WA, USA

Zachary Garrett ZACHGARRETT@GOOGLE.COM
Google Research
Seattle, WA, USA

Editor: Dan Alistarh

Abstract

Federated learning (FL) is a framework for learning across an axis of group partitioned data (heterogeneous clients) while preserving data privacy, under the orchestration of a central server. FL methods often compute gradients of loss functions purely locally (e.g. at each client), typically using automatic differentiation (AD) techniques. In this work, we consider the problem of applying AD to federated computations while preserving compatibility with privacy-enhancing technologies. We propose a framework, federated automatic differentiation (federated AD), that 1) enables computing derivatives of functions involving client and server computation as well as communication between them and 2) operates in a manner compatible with existing federated technology. We show, in analogy with AD, that federated AD may be implemented using various accumulation modes, which introduce distinct computation-communication trade-offs and systems requirements. Further, we show that a broad class of federated computations is *closed* under these modes of federated AD, implying that if the original computation can be implemented using privacy-preserving primitives, its derivative may be computed using the same primitives. We then show how federated AD can be used to create algorithms that dynamically learn components of the algorithm itself. We demonstrate that performance of FedAvg-style algorithms can be significantly improved by using federated AD in this manner.

Keywords: Federated learning, automatic differentiation, differentiable programming

1. Introduction

Federated learning (FL) is a paradigm for distributed learning in which clients collaboratively learn without directly sharing data. When combined with formal privacy mechanisms, it enables learning across decentralized data sources while preserving the privacy of the clients. FL encompasses a variety of settings with widely varying system characteristics. Two particularly noteworthy settings are *cross-device FL*, characterized by its many lightweight, unreliable clients, and *cross-silo FL*, characterized by a moderate number of reliable clients. We defer to Kairouz et al. (2021) for a more detailed introduction and taxonomy.

FL has seen notable success in production systems and applications (Hard et al., 2018; Yang et al., 2018; Bonawitz et al., 2019; Huba et al., 2022; Paulik et al., 2021; Hard et al., 2020; Chen et al., 2019; Ramaswamy et al., 2019). Despite this progress, developing and deploying effective FL methods remains difficult. When data is heterogeneous, FL methods can fail to converge to critical points of the underlying loss, or may not converge at all (Malinovskiy et al., 2020; Pathak and Wainwright, 2020). FL methods often use multiple local training steps per communication round to reduce total communication (McMahan et al., 2017), which can dictate a trade-off between initial convergence and accuracy (Charles and Konečný, 2021). This algorithmic pattern means that many FL methods are not equivalent to (stochastic) gradient descent on any loss function (Charles and Rush, 2022). These and related observations have inspired a slew of work that attempts to improve convergence via techniques such as learning rate decay (Hou et al., 2021) and control variates used to mitigate "client drift" (Karimireddy et al., 2020; Mitra et al., 2021; Mishchenko et al., 2022).

Unfortunately, such methods often require significant theoretical insight to design, may not work in cross-device FL settings (Mishchenko et al., 2022), fail to perform well in practice (Mitra et al., 2021), or may be incompatible with formal privacy techniques such as differential privacy and secure aggregation, which can limit what algorithms are possible to execute (see Kairouz et al. 2021, Chapter 4). They can also be difficult to tune. Hyperparameter tuning in FL can be prohibitively complex due to things like data access restrictions, system constraints, and the presence of many hyperparameters (Khodak et al., 2021). Moreover, as discussed above, FL methods that use fixed hyperparameters may face convergence issues regardless of tuning (Charles and Konečný, 2021).

In this work, we consider the problem of how to make it easier to design, implement, and dynamically adjust FL algorithms and their hyperparameters during training. We are motivated by the success of automatic differentiation (AD) in enabling rapid development of new algorithms and techniques in machine learning (see (Baydin et al., 2018b) for a thorough survey). The development of ML-focused AD frameworks, such as JAX (Bradbury et al., 2018), TensorFlow (Abadi et al., 2016), and PyTorch (Paszke et al., 2019), has accelerated this, enabling easy differentiation of loss functions with respect to user-specified parameters. We wish to enable such ease-of-use functionality and rapid experimentation in FL settings.

We are particularly interested in using AD to dynamically adjust FL algorithms in tandem with training. While such methods can fail due to perturbation confusion (Siskind and Pearlmutter, 2005; Manzyuk et al., 2019) and convergence issues (Christianson, 1994; Habiba and Pearlmutter, 2021), in practice they can often yield promising results when applied to neural network training. For example, AD has been applied to learned optimizers and update rules (Bengio et al., 1991; Schmidhuber, 1994; Andrychowicz et al., 2016; Wichrowska et al., 2017; Li and Malik, 2016, 2017; Lv et al., 2017; Bello et al., 2017; Metz et al., 2019b; Sandler et al., 2021; Vicol et al., 2021; Metz et al., 2022), neural architecture search (Elsken et al., 2019; Zoph and Le, 2017), meta-learning (Finn et al., 2017; Sandler et al., 2021; Metz et al., 2019a; Rusu et al., 2019; Pham et al., 2018), and learned compressors (Oktay et al., 2020). One noteworthy example of this dynamic adjustment of algorithms is *hypergradient descent* which applies (stochastic) gradient descent to optimizer hyperparameters, such as the learning rate, in tandem with training (Bengio, 2000; Baydin et al., 2018a).

Hypergradient descent has seen recent investigation in FL settings, due in part to the aforementioned difficulties of hyperparameter tuning. For example, the exponentiated gradient

descent mechanism proposed by (Khodak et al., 2021) is a form of numerical differentiation with respect to hyperparameters, Kan (2022) apply approximate hypergradient descent to certain hyperparameters, and Tarzanagh et al. (2022) use hypergradient approximations for a class of bi-level federated optimization problems. Wang et al. (2023) derive specific hypergradient descent algorithms for FL methods[1]. While these methods are theoretically-justified and empirically effective, they rely on manually-derived gradient formulas and approximations. The process of computing such quantities is often time-consuming, error-prone, and potentially inefficient from an implementation perspective (Baydin et al., 2018b).

1.1 Contributions

In this work, we seek to unify FL research with advanced AD by developing a framework for *federated automatic differentiation*, or federated AD. This framework enables the computation of *exact* gradients through general federated computations,[2] potentially involving multiple communication rounds between clients and server. Notably, this framework allows us to differentiate arbitrary outputs with respect to arbitrary (server-side) inputs, without relying on hand-computed derivative formulas, numerical derivatives, or gradient approximations. Like AD, it can be applied to computations involving programming constructs such as recursion and branching.[3]

Federated AD operates by augmenting a federated computation with a component that computes its derivative. Just as AD has different implementations (or "modes"), we give three primary modes for federated AD (forward-, reverse-, and mixed-mode). While the three modes have different system and communication overheads and requirements (which we discuss in detail), they enable exact gradient computation with only a constant factor of computation overhead. Moreover, we show that for a large class of computations, each of these modes preserves compatibility with formal privacy mechanisms, including differential privacy and secure aggregation.

After building up the federated AD framework, we show how it can be used to perform federated hypergradient descent in tandem with the popular FedOpt method (Reddi et al., 2021). In contrast to prior work (Khodak et al., 2021; Kan, 2022; Tarzanagh et al., 2022) federated AD enables applying hypergradient descent to both client hyperparameters and server hyperparameters. We then specialize this methodology to 1) learning server optimizer hyperparameters and 2) learning client weighting schemes. We apply the resulting methods to a variety of benchmark FL tasks. We find that the resulting method can exhibit significantly better convergence properties than "static" FedOpt, and its ability to dynamically adjust the hyperparameters over time plays a crucial role in this behavior.

1.2 Goals and Limitations

Our goal is to begin the development of a framework and set of tools that makes it easier to develop dynamic algorithms for FL. While we apply federated AD to derive algorithms for

1. We note that this appeared after the first version of this work.
2. In short, these are computations on data that have an explicit notion of where the data resides in a federated system. We will discuss these in detail in Section 2.2.
3. Since the initial version of this work appeared, we have developed an open-source library that implements federated AD in JAX. See (Rush et al., 2024) for details.

federated hypergradient descent, we stress that this is a relatively simple use of federated AD. We focus on 1) showcasing the utility of federated AD in enabling dynamically learned parameters and 2) showing how federated AD enables powerful new classes of FL methods. We do not attempt to demonstrate that this application of federated AD outperforms related methods (such as the methods of (Wang et al., 2023)). Rather, we attempt to motivate, sketch, and enable research in increased dynamism for FL algorithms by the introduction of a new tool in the FL researcher's repertoire. This framework also enables a substantial simplification of many prior methods, for example by reducing the need for gradient approximations. By way of analogy, backpropagation has served as a fundamental enabler to the dramatic development of centralized ML by enabling efficient and flexible derivative computations, while also simplifying model and algorithm expression. We seek to mirror this in the younger field of FL.

2. Preliminaries and Background

Throughout this work, we assume that all relevant functions are differentiable. The results all hold with sub-differentiable functions as well, but we restrict to differentiability for notational simplicity. In the remainder of this section, we provide sufficient background on various topics in order to give a fully-featured but succinct description of federated AD. We do not attempt to give a complete overview of topics, and give references as needed.

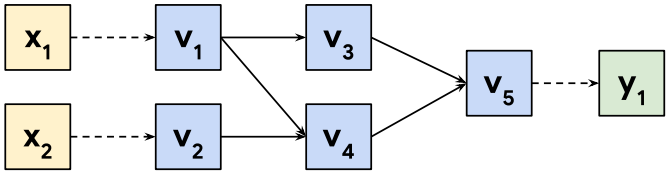
2.1 Automatic Differentiation

Automatic differentiation (AD), also known as "autodiff", is a family of techniques used to efficiently compute derivatives of numeric functions expressed as computer programs. While we defer the interested reader to works such as (Baydin et al., 2018b) for a more complete picture, we will discuss some background on AD that will serve as grounding for the remainder of this work.

AD methods typically augment function evaluation with auxiliary derivative computations. This is done by reducing programs to a set of "elementary" operators whose derivatives are known, and applying the chain rule to accumulate and combine these derivatives. Because this is done by decomposing the program itself, it can be applied to functions involving complex programming constructs, leveraging the observation that "AD is blind with respect to any operation, including control flow statements, which do not alter numeric values" (Baydin et al., 2018b, Section 3).

We can represent functions as **computational graphs** (Bauer, 1974). Using the notation of Griewank and Walther (2008), we represent the input variables as $x_1, \ldots, x_n$ and the outputs as $y_1, \ldots, y_m$. The inputs and outputs are linked by a directed acyclic graph (DAG) with intermediate nodes $v_1, \ldots, v_l$, where each intermediate node is some elementary function of its parents in the graph.

Figure 1: Computational graph of $f(x_1, x_2) = \sin(x_1) + x_1 x_2$. Here, v_i is an intermediate value of the computation as given in (1).



For example, $y_1 = \sin(x_1) + x_1 x_2$ can be computed via elementary operations as follows:

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= \sin(v_1) \\ v_4 &= v_1 v_2 \\ v_5 &= v_3 + v_4 \\ y_1 &= v_5 \end{aligned} \tag{1}$$

This has a computational graph given in Fig. 1. Note that the dashed lines here indicate equality (that is, $v_1 = x_1, v_2 = x_2, v_5 = y_1$). Suppose we wish to compute the derivative of y_1 with respect to x_1. Two standard AD techniques for this are **forward-mode** and **reverse-mode** auto-differentiation. In forward-mode, we move forward in the graph starting from x_1. As we compute each intermediate node v_i, we also compute

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} \tag{2}$$

via the chain rule, using the fact that we know v_j, $\dot{v}_j$ for $j < i$. For example, applying the chain rule to (1),

$$\dot{v}_5 = \frac{\partial (v_3 + v_4)}{\partial x_1} = \dot{v}_3 + \dot{v}_4.$$

In reverse-mode, we first compute a "forward pass" through the graph to compute all intermediate v_i. We then move backwards through the graph, starting at y_1. At each node we compute

$$\overline{v}_i = \frac{\partial y_1}{\partial v_i} \tag{3}$$

using the chain rule, using the fact that we know v_j, $\overline{v}_j$ for $j > i$. For example, applying the chain rule to (1),

$$\begin{aligned} \overline{v}_1 &= \frac{\partial y_1}{\partial v_1} \\ &= \frac{\partial y_1}{\partial v_3}\frac{\partial v_3}{\partial v_1} + \frac{\partial y_1}{\partial v_4}\frac{\partial v_4}{\partial v_1} \\ &= \overline{v}_3 \cos(v_1) + \overline{v}_4 \frac{v_4}{v_1}. \end{aligned}$$

Both methods generalize to compute the Jacobian of a vector-valued function. The difference between them corresponds to choosing an order for multiplications of Jacobian matrices specified by the chain rule, with 'forward-mode' corresponding to left-to-right multiplication and 'backwards mode' right-to-left. If $f : \mathbb{R}^n \to \mathbb{R}^m$, then forward-mode requires n forward passes to compute this Jacobian (one for each input), while reverse-mode requires m backwards passes (one for each output). This observation is the root of preference for reverse-mode accumulation while training machine learning models (known as 'backpropagation' in this special case); we differentiate a scalar-valued function with respect to a high-dimensional parameter vector (that is, $m = 1, n \gg 1$).

2.2 Federated Computations

Our work focuses on taking derivatives of distributed functions, particularly functions whose computation spans clients and server in FL settings. In order to formalize this, we will use the concepts of **federated values** and **federated computations** as defined in the TensorFlow Federated framework (Ingerman and Ostrowski, 2019), and discussed in some detail by Charles et al. (2022). A **federated value** is some data hosted across a group of devices in a distributed system. Notably, it has a value and a *placement* (that is, where is it located). [4] We focus on two placements:

1. **Server-placed values**. Conceptually, these are singleton values representing data directly accessibly by the server within a federated computation. While our work can be generalized to settings with multiple servers, we will assume throughout that there is a single server.

2. **Client-placed values**. Conceptually, these are multiple values representing the data accessible by each client within a federated computation. We assume a form of semantic symmetry, so that if one client has a type of data (for example, a local set of model weights) then so too do all other clients participating in the federated computation. We model the collection of values across all participating clients (a set we denote by C) as a single federated value.

When useful, we use x`@SERVER` to denote a value of x placed at the server, and $\{x_i \mid i \in C\}$`@CLIENTS` to denote a client-placed federated value, where each participating client $i \in C$ has a corresponding value x_i. We often abbreviate the latter by $\{x_i\}$`@CLIENTS` or even x`@CLIENTS` (with the values x_i implicit).

A **federated computation** is simply a function whose inputs and outputs are federated values. For example, a commonly-used federated computation is `federated_broadcast`. This is simply the function that sends a server-placed value to all participating clients, ie:

$$\texttt{federated\_broadcast}\,(x\texttt{@SERVER}) = x\texttt{@CLIENTS}$$

4. The concept of placement in TensorFlow Federated is one of logical partition. It does not necessarily indicate the physical location of the device. While we use the placement and the physical location of data interdependently in our work, we note that placement can serve as a more general organizing mechanism (for example, by having multiple `@CLIENTS` placements that denote different sets of clients).

where each client i has the same value x. Another important federated computation is a *federated sum*. This is simply the function given by:

$$\texttt{federated\_sum}\,(x\texttt{@CLIENTS}) = \left(\sum_{i \in C} x_i\right) \texttt{@SERVER}.$$

Intuitively, this represents summing up values across all clients participating in a given round.

Given a (non-federated) function g and placed values $x\texttt{@SERVER}$ and $z\texttt{@CLIENTS}$ (representing client values $\{z_i \mid i \in C\}$), we define

$$g(x\texttt{@SERVER}) := g(x)\texttt{@SERVER}, \tag{4}$$

$$g(z\texttt{@CLIENTS}) := \{g(z_i) \mid i \in C\}\texttt{@CLIENTS}. \tag{5}$$

In other words, we can form a federated computation from g by applying it to local data (either at the server, or at each client).

FL algorithms are often designed with explicit *data minimization* principles. This is typically done by preventing clients from directly sharing their own data with any other agent in the system. However, FL algorithms must be augmented with explicit privacy mechanisms if formal privacy guarantees are desired; see (Kairouz et al., 2021) for detail on core mechanisms like differentially private aggregations and cryptographically secure aggregation (SecAgg, Bonawitz et al. (2017)). We do not go into detail on these privacy-preserving mechanisms, except to note that many such mechanisms are often incompatible with server→client communication that is client-dependent or client→server communication that is not sum-based (such as median-based aggregation schemes).[5] Thus, in privacy-sensitive settings, we generally want to restrict to federated computations that only use `federated_broadcast` and `federated_sum`, in order to ensure compatibility with a wide array of privacy mechanisms. This precludes the use of things like client-dependent broadcast mechanisms (such as `federated_select` (Charles et al., 2022)) or median-based aggregation (Yin et al., 2018). While such communication primitives are not necessarily incompatible with formal privacy, they often require more specialized privacy mechanisms, and may depend strongly on the implementation (Charles et al., 2022).

2.3 Motivating Questions

We are now ready to state the primary questions motivating our work.

Question 1 *How do we differentiate through federated computations?*

There are naive answers to this that are impractical or lack compatibility with privacy-preserving technologies. For example, suppose we have a federated computation

$$f(x\texttt{@SERVER}, y\texttt{@CLIENTS}) = z\texttt{@SERVER}$$

and that we wish to compute the derivative of z with respect to x. A non-private way to do this would be to implement the function f in a single-process fashion; The clients could send

5. For details on these restrictions, see (Kairouz et al., 2021, Section 4), especially the discussion of SecAgg and the shuffle model of differential privacy. See (Pillutla et al., 2022) for an informative example of designing SecAgg-compatible robust aggregation methods.

their data to the server, and the server could then compute the derivative of z with respect to x using AD. This approach clearly violates data-minimization principles of FL. Further, requiring a single process to compute and differentiate a function which accepts an input per-client could introduce scalability limitations; if the number of clients is large and the size of each constituent y_i of y`@CLIENTS` is also large, it may be infeasible for the server to directly compute z, let alone its derivative.

We would like to ensure that the server does not have direct access to client data, and that server-side computation can be implemented in a manner compatible with modern distributed systems. Therefore, we may wish to use privacy-preserving distributed technologies and existing distributed federated computation infrastructure to compute both z and its derivative with respect to x. Thus, what we are actually interested in is the following:

Question 2 *How do we differentiate through federated computations in a way that is both scalable and compatible with privacy-preserving technologies?*

We give a partial answer to this in Section 3. In particular, we develop a framework, federated automatic differentiation (federated AD) that enables us to compute derivatives for broad classes of federated computations. Moreover, we show that if a federated computation f is compatible with certain privacy mechanisms (including differential privacy and secure aggregation), then we can use federated AD to compute this kind of derivative of f while maintaining compatibility with secure aggregation and differential privacy.[6]

While machine learning literature generally suggests that computing derivatives is important, it is not necessarily clear how one could put derivatives of federated computations to good use. Thus, after discussing Question 2, we give a partial answer to the following question:

Question 3 *How can we use federated automatic differentiation to design improved federated learning algorithms?*

We discuss this question in Section 4, to show that federated AD can be used to develop self-tuning federated optimization algorithms. In Section 5 and Section 6, we show that these self-tuning algorithms can yield improved performance over benchmark methods for federated optimization. We believe that federated AD can enable significant work in the design of dynamic federated algorithms.

3. Federated Automatic Differentiation

In order to consider Question 2, we begin by specializing the derivatives we compute. In the following, we differentiate with respect to server-placed parameters.[7] In practice, this restriction is likely to be minimal. Server-placed parameters are the traditional entry point by which a modeler or algorithm designer may specify behavior of a federated computation.

6. We will discuss several 'modes' of federated AD, analogous to forward- and reverse-mode in traditional AD. This claim is true for *all* federated computations in forward-mode, and all computations which use summation to aggregate across clients in reverse-mode.
7. Our framework can be applied analogously to values that are constant across clients, but we restrict to server-placed values for simplicity of presentation. Note that we can model values that are constant across clients as the result of a call to `federated_broadcast`.

We illustrate our methods by considering a specific class of federated computations, though we note that federated AD may be applied more generally. While relatively simple, this class illustrates our core approach for differentiating through federated computations.

Class 1 *We assume that the server has some value, x, and each client i has their local data z_i. We first consider federated computations of the form*

$$f(x\texttt{@SERVER}, z\texttt{@CLIENTS}) = y\texttt{@SERVER}$$

that involve a single call to `federated_broadcast` *and* `federated_sum`*. For simplicity, we assume f is defined as follows:*

1. *Server holds $v_1\texttt{@SERVER} = x\texttt{@SERVER}$.*
2. *Server computes $v_2\texttt{@SERVER} = f_1(v_1\texttt{@SERVER})$.*
3. *Clients receive $v_3\texttt{@CLIENTS} = \texttt{federated\_broadcast}(v_2\texttt{@SERVER})$*
4. *Clients compute $v_4\texttt{@CLIENTS} = f_2(v_3\texttt{@CLIENTS}, z\texttt{@CLIENTS})$.*
5. *Server receives $v_5\texttt{@SERVER} = \texttt{federated\_sum}(v_4\texttt{@CLIENTS})$.*
6. *Server computes $y\texttt{@SERVER} = v_6\texttt{@SERVER} = f_3(v_5\texttt{@SERVER})$.*

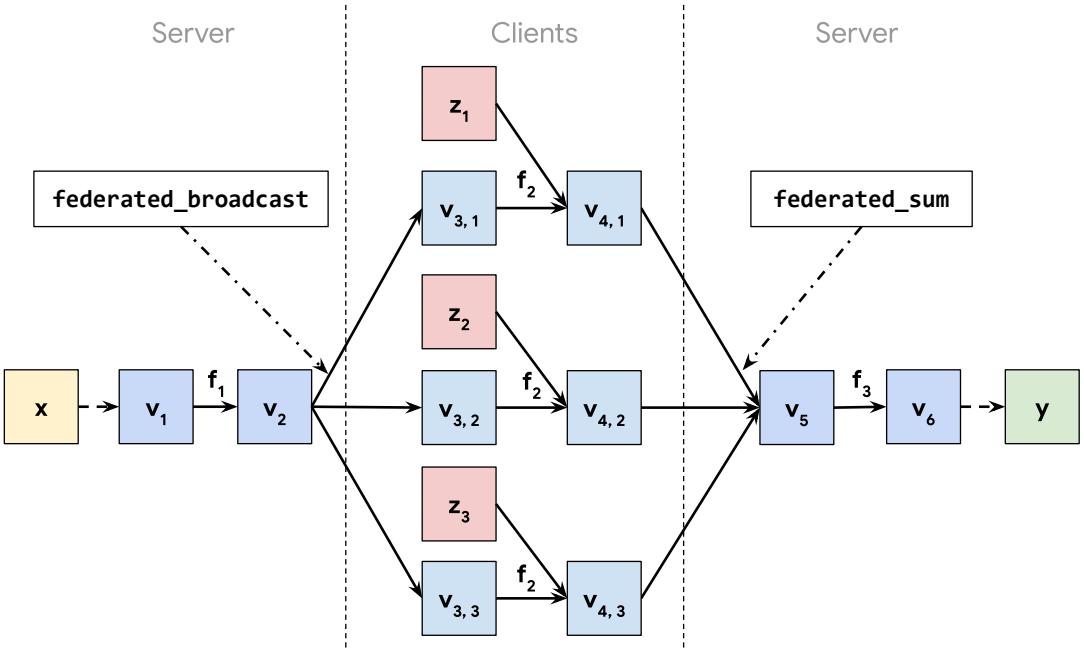
This class of functions includes the per-round logic of many notable algorithms, including FedAvg (McMahan et al., 2017), FedProx (Li et al., 2020), PerFedAvg (Fallah et al., 2020), FedOpt (Reddi et al., 2021), and many more.

We wish to compute $\partial f/\partial x$ without sharing the local client data z_i. To understand how to do this, we first visualize f as a **federated computational graph**. This is analogous to the computational graph in Section 2.1, but where each value in the graph is a federated value (see Section 2.2), where nodes are augmented with placements (server or clients). This is pictured in Fig. 2.

Ignoring data placement and assuming that the number of clients per round remains fixed, we could simply treat Fig. 2 as a non-federated computational graph and use AD techniques. However, care must be taken when crossing communication boundaries so as not to violate data minimization principles of FL and to ensure compatibility with formal privacy mechanisms. Additionally, in many FL settings we would like to restrict the amount of communication incurred (and further may need to take into consideration asymmetry between download and upload bandwidth).

Below we present algorithms for **federated automatic differentiation** (federated AD) that meet the above criteria. Notably, we give forward-mode, reverse-mode, and mixed-mode federated AD algorithms, and show how they can be viewed as augmentations and transformations of federated communication patterns. Each of the approaches sketched below has distinct systems characteristics and requirements, as well as different amounts of communication incurred, as discussed in Section 3.4.

Figure 2: Federated computational graph for $y = f(x)$ as in Class 1, where three clients participate. Here, v_i represent intermediate values used in the computation. Each value has a placement (server or clients), and the clients have data z_i which are not shared with one another or the server. Server→client communication is done via `federated_broadcast`, and client→server communication is done via `federated_sum`.



3.1 Forward-Mode Federated AD

We begin with forward-mode federated AD due to its simpler presentation. The crucial observation for applying forward-mode federated AD to a federated computation f is that it can be implemented using the same communication pattern as the computation of f. In particular, this means that we can use the same primitives (e.g. `federated_broadcast` and `federated_sum`), but applied to forward-mode derivatives, in order to compute the desired derivative, without sharing the local data z_i.

We illustrate forward-mode federated AD by stepping through differentiation of the function class described in Class 1. As in Section 2.1, let $\dot{v}_i$ denote the Jacobian of v_i with respect to x. We then have the following straightforward observations, consequence of the fact that derivatives commute with sums.[8]

$$\dot{v}_3\texttt{@CLIENTS} = \texttt{federated\_broadcast}(\dot{v}_2\texttt{@SERVER}) \tag{6}$$

$$\dot{v}_5\texttt{@SERVER} = \texttt{federated\_sum}(\dot{v}_4\texttt{@CLIENTS}) \tag{7}$$

8. Here and in Eqs. (8) and (9) we rely on the fact that we specialize to differentiating with respect to a single, server-placed values. Differentiating with respect to a clients-placed value would instead result in multiple values (one per client). For example, one could argue that the correct derivative of a sum with respect to its clients-placed argument is the value `1@CLIENTS`.

Table 1: Forward mode federated AD applied to f in the class of functions described in Class 1. We compare the procedure for computing $f(x)$ (left) with the augmented computation that also computes $\partial f/\partial x$ via forward mode federated AD (right), with federated AD-specific components marked in blue. Note that x is a server-placed argument. We use $\dot{v}$ to denote $\partial v/\partial x$ (see Section 2.1).

Evaluation of $y = f(x)$	Forward mode evaluation of $y = f(x)$ and $\dot{y} = \partial f/\partial x$
Server receives input $v_1 = x$	Server receives input $v_1 = x$
Server computes v_2	Server computes v_2 and $\dot{v}_2$
Clients receive:	Clients receive:
$v_3 =$ `federated_broadcast`(v_2)	$v_3, \dot{v}_3 =$ `federated_broadcast`$(v_2, \dot{v}_2)$
Clients compute v_4	Clients compute v_4 and $\dot{v}_4$
Server receives $v_5 =$ `federated_sum`(v_4)	Server receives $v_5, \dot{v}_5 =$ `federated_sum`$(v_4, \dot{v}_4)$
Server computes v_6	Server computes v_6 and $\dot{v}_6$
Server outputs $y = v_6$	Server outputs $y = v_6$ and $\dot{y} = \dot{v}_6$

According to the federated computational graph in Fig. 2, all that client i needs to compute $\dot{v}_{4,i}$ are $z_i, v_{3,i}, \dot{v_{3,i}}$, which are available to the client, and $\dot{v}_{3,i}$, which they can receive via `federated_broadcast`$(\dot{v}_2$`@SERVER`$)$ as per (6). Thus, if the server computes $\dot{v}_2$, then the clients can compute $\dot{v}_{4,i}$. Similarly, for the server to compute $\dot{v}_6$, it requires v_5 and $\dot{v}_5$. This last quantity is simply `federated_sum`$(\dot{v}_4$`@CLIENTS`$)$ as per (7).

This implies that forward-mode federated auto-differentiation can be done by following the same procedure as in Class 1, but applying it to both the function values v_j and derivatives $\dot{v}_j$. A side-by-side comparison of computing $f(x)$ versus computing $f(x)$ and $\partial f/\partial x$ via forward-mode federated AD is given in Table 1. Our presentation here is drawn from (Baydin et al., 2018b, Table 2), in part due to our admiration for the authors' wonderful exposition, and in part to demonstrate the conceptual similarities between forward-mode federated AD and forward-mode AD.

Each of Eqs. (6) and (7) illustrates an important principle: When performing federated AD, derivatives must necessarily be communicated across device boundaries. Since communication bandwidth is often a critical limitation in federated systems, the size and shape of these derivatives may be crucial to the feasibility of computing the derivative of a particular federated computation. As we will see below, analogously to centralized AD, the size of the derivatives which flow across communication boundaries may be significantly altered by the mode in which we compute federated AD.

3.2 Reverse-Mode Federated AD

In centralized AD, particularly for machine learning applications, reverse-mode accumulation has obvious operational advantages to forward accumulation. Reverse-accumulation mode may also be defined in the federated setting, where, in direct analogy with centralized AD, it requires both re-addressing the same set of clients and reversing the communication patterns.

Letting $\overline{v}_i$ denote $\partial y/\partial v_j$, we have the following observations about Fig. 2:

$$\overline{v}_2\texttt{@SERVER} = \texttt{federated\_sum}(\overline{v}_3\texttt{@CLIENTS}) \tag{8}$$

Table 2: An example of reverse-mode federated AD when applied to the class of functions described in Class 1. For comparison, we give a basic algorithm for computing $f(x)$ (left), as well as a backwards computation that also computes $\partial f/\partial x$ via reverse-mode federated AD (right). Note that the reverse-mode computations depend on access to the previously computed values in the forward pass. We use $\overline{v}$ to denote $\partial y/\partial v$ (see Section 2.1).

Evaluation of $y = f(x)$	Reverse mode evaluation of $\overline{x} = \partial f/\partial x$
Server receives input $v_1 = x$	Server outputs $\overline{x} = \overline{v}_1$
Server computes v_2	Server computes $\overline{v}_1$
Clients receive $v_3 =$ `federated_broadcast`(v_2)	Server receives $\overline{v}_2 =$ `federated_sum`$(\overline{v}_3)$
Clients compute v_4	Clients compute $\overline{v}_3$
Server receives $v_5 =$ `federated_sum`(v_4)	Clients receive $\overline{v}_4 =$ `federated_broadcast`$(\overline{v}_5)$
Server computes v_6	Server computes $\overline{v}_5$
Server outputs $y = v_6$	Server receives input $\overline{v}_6 = \overline{y}$

$$\overline{v}_4\texttt{@CLIENTS} = \texttt{federated\_broadcast}(\overline{v}_5\texttt{@SERVER}) \tag{9}$$

These mirror (6) and (7) for forward-mode federated AD, but reverse the communication primitives involved. This gives us a straightforward way to perform reverse-mode federated AD. We first do a forward pass over the computation of $f(x)$, following the procedure in Class 1. Mirroring Section 2.1, we use a backwards pass to compute $\overline{x}$. To do this, we reverse the arrows in Fig. 2 and traverse the graph in reverse, starting at y. Whenever we encounter a value v_j, we use the available data from the forward-pass and previous nodes in the graph to compute $\overline{v}_j$. The only missing component is how to reverse the communication primitives used (`federated_sum` and `federated_broadcast`). Eq. (8) and Eq. (9) imply that we simply swap the two. For example, the computation of $v_5\texttt{@SERVER} = \texttt{federated\_sum}(v_4\texttt{@CLIENTS})$ in the forward pass of Class 1 becomes $\overline{v}_4\texttt{@CLIENTS} = \texttt{federated\_broadcast}(\overline{v}_5\texttt{@SERVER})$ in reverse-mode. Pseudo-code for this reverse-mode traversal is given in Table 2.

Notably, reverse-mode federated AD only uses `federated_sum` and `federated_broadcast` to perform communication, and does not involve directly sharing client data. It is therefore compatible with the same privacy-preserving mechanisms as forward-mode federated AD. It also shares some of the advantages of reverse-mode centralized AD, as discussed in Section 3.4.

3.3 Mixed-Mode Federated AD

In non-federated settings, forward-mode and reverse-mode AD can both be viewed as traversals of a computational graph, where the derivative is accumulated as we traverse. Notably, these two modes are in some sense "extreme": we either start at a root of the corresponding DAG and only move forward, or start at a leaf and only move backwards. However, as long as we correctly apply the chain rule, we have freedom to traverse the graph in all manner of ways (e.g. moving forward across some edges, and backwards across others). In fact, such "mixed-mode" accumulation of derivatives can result in a smaller number of required arithmetic operations (though finding the minimum number of arithmetic operations required is NP-complete) (Naumann, 2008).

Table 3: Mixed-mode federated AD applied to f in the class of functions described in Class 1. We compare the procedure for computing $f(x)$ (left) with the mixed-mode federated AD computations (right). While the communication pattern allows mixed-mode federated AD to be done in tandem with the forward-pass, derivatives are computed locally by the server or clients using reverse-mode (not federated) AD.

Evaluation of $y = f(x)$	Mixed-mode evaluation of $\partial y / \partial x$
Server receives input $v_1 = x$ Server computes v_2	Server computes $\frac{\partial v_2}{\partial v_1}$ via reverse-mode AD
Clients receive $v_3 = \texttt{federated\_broadcast}(v_2)$ Clients compute v_4	Clients compute $\frac{\partial v_4}{\partial v_3}$ via reverse-mode AD
Server receives $v_5 = \texttt{federated\_sum}(v_4)$	Server receives $\frac{\partial v_5}{\partial v_2} = \texttt{federated\_sum}\left(\frac{\partial v_4}{\partial v_3}\right)$
Server computes v_6	Server computes $\frac{\partial v_6}{\partial v_5}$ via reverse-mode AD
Server outputs $y = v_6$	Server outputs $\frac{\partial y}{\partial x} = \frac{\partial v_6}{\partial v_1} = \frac{\partial v_6}{\partial v_5}\frac{\partial v_5}{\partial v_2}\frac{\partial v_2}{\partial v_1}$

Mixed-mode differentiation is attractive in the federated setting, as different devices may face different limits on computation and communication, and there may be asymmetry in cost between different types of communication (e.g. up-link vs down-link). In general, we often wish to minimize communication to and from clients, and minimize the total amount of computation on the client. We may also wish to remove the systems issues which hinder implementation of reverse-mode federated AD, where clients dropping out between the forward-pass and reverse-pass can lead to bias in the gradient computation.

We can do this by decoupling the overall traversal of the federated computational graph in Fig. 2 from the local graph traversal done by the server or clients. Specifically, we can use the chain rule to move forward through the communication primitives used (the `federated_broadcast` and `federated_sum` calls, in that order) but use reverse-mode locally at each device (for example, we can use reverse-mode for the part of the computation that occurs solely at the clients). The resulting procedure is given in Table 3.

In comparison to forward-mode federated AD, we see mixed-mode does not require sending (potentially large) Jacobian matrices to the clients, and allows clients to use reverse-mode AD locally. In comparison to reverse-mode federated AD, we again see that no extra server→client communication is necessary, and that the derivative computations can be done in tandem with the forward pass.

Just as forward- and reverse-mode federated AD can be extended to larger classes of computations, so too can mixed-mode federated AD, using similar principles. In general, mixed-mode federated AD can be applied to any computation whose communication is performed via `federated_broadcast` and `federated_sum`, and the derivative computation itself only requires `federated_sum`. As a result, mixed-mode federated AD is also compatible with formal privacy mechanisms. Finally, alternate specifications of mixed-mode federated

AD are possible, which may be more or less desirable based on the particulars of the implementing system and the computation being differentiated.

3.4 System Considerations

We concretize an example from Class 1 and walk through differentiation in the three modes discussed above in order to highlight the systems and communication considerations at play when choosing a federated AD mode.

Suppose that the function f_1 of Class 1 (which runs on the server before the model broadcast) performs a *distillation* of the incoming server-side model x. That is, suppose x represents a vector of dimensionality n and $f_1(x)$ computes a vector u of dimensionality m, where $m \ll n$. The function f_2 computes the loss of a client on the distilled model, and the function f_3 is simply the identity function. Thus, the remainder of our computation simply broadcasts and evaluates the distilled model u across the clients before computing an average of the losses on the server, which we denote by y. This computation broadcasts m floats and aggregates a single float from each client.

Now, suppose we wish to differentiate y with respect to x. In **forward-mode**, we compute and communicate $\frac{du}{dx}$, which in our case (assuming matrices act on the left) is a matrix $\mathbf{A}$ of shape $m \times n$. We broadcast this (potentially quite large) matrix to the clients, which locally compute the derivative of their local losses with respect to their incoming models, which may be represented as matrices $\mathbf{B}$ of shape $1 \times m$. Clients must then perform the multiplication $\mathbf{BA}$, resulting in each client owning a matrix of shape $1 \times n$, which will then be averaged. Note that the floats broadcasted and aggregated in forward-mode are both inflated by a multiplicative factor of n, the dimensionality of the value with respect to which we differentiate.

In **reverse-mode**, at the end of the 'forward pass' of the computation, the server broadcasts a *scalar* value to each client. Clients then compute $\frac{dy}{du}$, of shape $1 \times m$. This value is then summed by the server and leveraged to compute $\frac{dy}{dx}$, a matrix of shape $1 \times n$, via local reverse-accumulation mode—in particular, without ever materializing the matrix $\mathbf{A}$ above.

In **mixed-mode**, we might pin all the matrix multiplies to the server, asking clients to compute the $1 \times m$ matrix representing derivative of local loss with respect to the broadcast model u. The server maintains (rather than broadcasts) the matrix state $\mathbf{A}$ of shape $m \times n$ (or the capacity to compute matrix-vector products with respect to $\mathbf{A}$), and performs a post-aggregation chain-rule multiplication with the aggregated client-side derivatives of shape $1 \times m$ to compute the derivative $\frac{dy}{dx}$ of shape $1 \times n$.

Several interesting features of these various implementations come immediately to light. Each of these modes has distinct computation and communication properties. Mixed mode actually communicates the *least*, though requires materializing the large matrix $\mathbf{A}$ (or the capacity to peform ex-post-facto vector-Jacobian products, effectively splitting up the implementation of the chain rule). Still, it can be implemented without assuming stable client connections. Reverse mode, despite its appealing communication and computation properties, requires this assumption, for the same reason that intermediate values must be stored (or recomputed) while performing backpropagation in centralized ML. Many existing cross-device FL systems are MapReduce-like in their structure (Bonawitz et al.,

2019), *designed without the ability to re-address the same cohort of clients.* This means that such systems are incompatible with the kind of reverse-mode federated AD computations in Table 2. Finally, though forward-mode *significantly* increases the communication costs of both upload and download, in some sense this is an artifact of the chosen example: forward-mode differentiating a vector-valued function with respect to a scalar parameter would show forward-mode federated AD to be preferable to reverse-mode.

Observations about the systems characteristics of the various modes of federated AD may be generalized to yield statements about the possibility of communication-efficient implementations. Preference for one implementation over another is potentially dependent on both the inputs and outputs of the function being differentiated as well as the cost model of the targeted federated system (e.g., as illustrated above, relative cost between upload and download may affect choice of forward versus reverse-mode).

4. Applying Federated Automatic Differentiation to Federated Learning

We now turn our attention to applying federated AD to FL tasks, particularly federated optimization. Throughout, we do not focus on creating "state-of-the-art" methods for federated optimization. Instead, we intend to demonstrate that federated AD can enable more expressive algorithms in which components of the federated computation are learned throughout training, and to good empirical effect.

While one can apply federated AD to a wide array of federated computations, for concreteness we will focus on applying it to learn portions of the FedOpt algorithmic framework (Reddi et al., 2021). FedOpt is an empirically successful and widely adopted federated optimization paradigm that combines client-level and server-level optimization.

4.1 FedOpt Framework

FedOpt works as follows: At each round t, the server has a model x_t. This model is broadcast to some set C_t of participating clients (via `federated_broadcast`). Each client $i \in C_t$ computes an updated model $x_{t,i}$ using some procedure ClientUpdate (typically some number of first-order optimization steps) with the client's local data z_i. The client then computes $\Delta_{t,i} = x_t - x_{t,i}$. The server receives the average Δ_t of these $\Delta_{t,i}$ from the clients, and uses this to update the server model x_t via some procedure ServerUpdate (typically a first-order optimization step).

Pseudo-code for FedOpt is given in Algorithm 1. In this algorithm, the server receives a mean of the client "model deltas" $\Delta_{t,i}$ weighted (respectively) by scalars ρ_i. We denote this server-placed federated value by $\texttt{federated\_mean}(\{\Delta_{t,i}\}\texttt{@CLIENTS}, \{\rho_i\}\texttt{@CLIENTS})$. Note that this operation can be reduced to `federated_sum` since

$$\texttt{federated\_mean}(\{\Delta_{t,i}\}\texttt{@CLIENTS}, \{\rho_i\}\texttt{@CLIENTS}) = \frac{\texttt{federated\_sum}(\{\rho_i \Delta_{t,i}\}\texttt{@CLIENTS})}{\texttt{federated\_sum}(\{\rho_i\}\texttt{@CLIENTS})}.$$

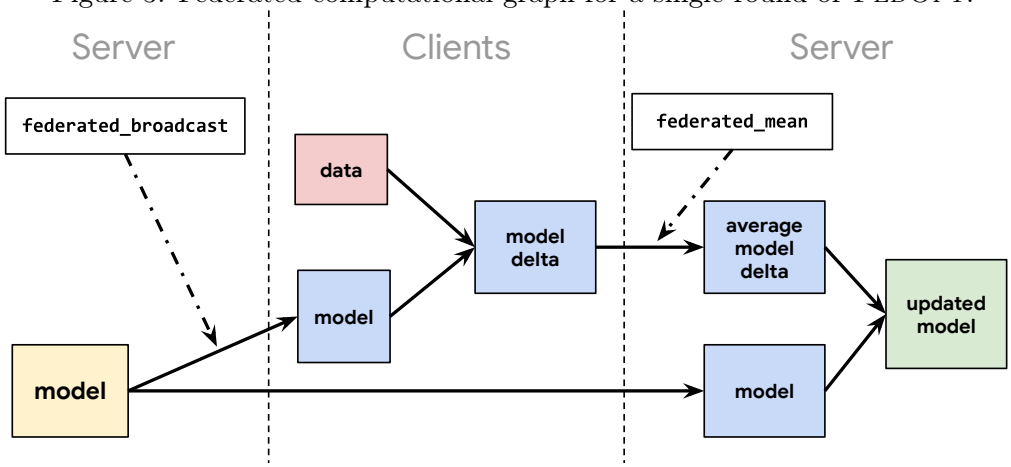
In particular, this means that the communication in FedOpt only involves `federated_broadcast` and `federated_sum`, and is therefore compatible with privacy-preserving technologies (e.g. SecAgg and differential privacy).

The client weights ρ_i are typically constants that are easily computed by each client. Two common weighting schemes are *example weighting*, where each client's weight is the number

Algorithm 1 FedOpt

Input: server weights x_0`@SERVER`, client data $\{z_i\}$`@CLIENTS`, client weights $\{\rho_i\}$`@CLIENTS`
for $t = 0, \cdots, T-1$ **do**
  Sample a cohort C_t of participating clients.
  Clients receive `federated_broadcast`$(x_t$`@SERVER`$)$
  for each client $i \in C_t$ **in parallel do**
    $x_{t,i} = \text{ClientUpdate}(x_t, z_i)$
    $\Delta_{t,i} = x_t - x_{t,i}$
  end for
  Server receives $\Delta_t =$ `federated_mean` $(\{\Delta_{t,i}\}$`@CLIENTS`$, \{\rho_i\}$`@CLIENTS`$)$
  Server computes $x_{t+1} = \text{ServerUpdate}(x_t, \Delta_t)$
end for

Figure 3: Federated computational graph for a single round of FedOpt.



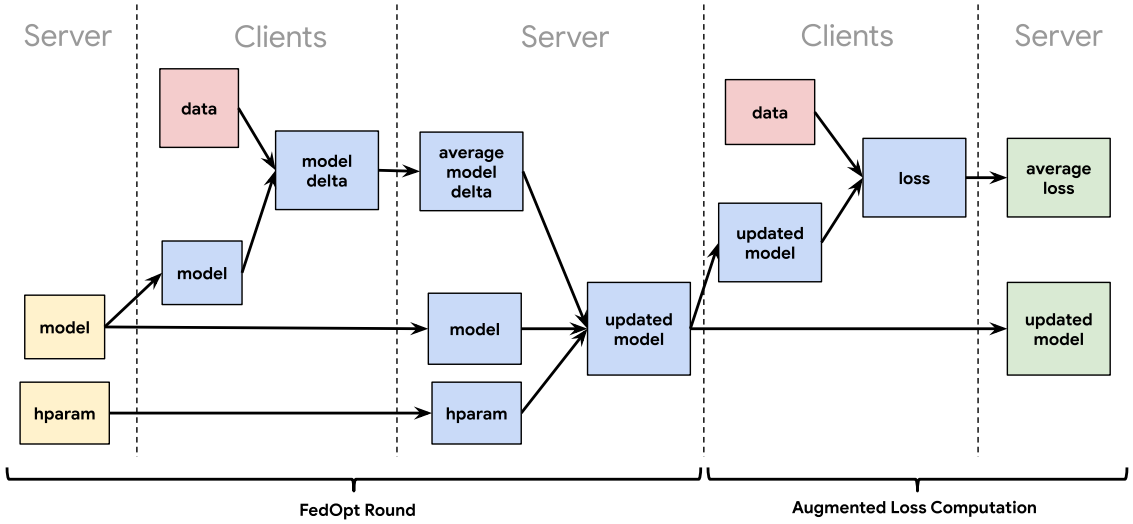
of examples they hold (so that $\rho_i = |z_i|$), and *uniform weighting*, where each client's weight is the same (so that $\rho_i = 1$). Example weighting was proposed by McMahan et al. (2017), and shown to be effective from an optimization standpoint by Li et al. (2020). However, uniform weighting is necessary for certain privacy mechanisms like differentially private averages, which are often unweighted to ease control of sensitivity.

For reference, we give a federated computational graph representing one round of FedOpt in Fig. 3. We only picture a single client, but conceptually the same client operations are being done in parallel on some set of participating clients (as in Fig. 2). We will return to Fig. 3 in later sections, in order to discuss applications of federated AD to FedOpt.

4.2 Federated Hypergradient Descent

As discussed in Section 1, we would like to enable FL algorithms that dynamically adjust their structure as they train. We now detail how to use federated AD to learn aspects of FedOpt in tandem with execution of federated AD by adjusting existing components of FedOpt. We do this by applying *federated hypergradient descent* to Algorithm 1. The primary obstacle to this is that individual clients can only compute hypergradients with respect to their own

Figure 4: Federated computational graph used to compute server hypergradients in FedOpt. All server→client communication is done via `federated_broadcast`, while all client→server is done via `federated_mean`. This computation produces some updated model and an estimate of the loss of that model.



loss function, while the server cannot directly compute any hypergradients (as it does not necessarily possess any data). Thus, in the absence of federated AD, it is not immediately clear how to compute derivatives of the form $\partial L(x)/\partial\alpha$ for some global loss function L at a model x with respect to some server hyperparameter α.

This roadblock was noted by prior works on federated hyperparameter tuning. While Khodak et al. (2021) use a sophisticated weight-sharing mechanism to learn client hyperparameters, they learn server hyperparameters by training multiple times with different hyperparameter settings and eliminating parameters that are not doing well (e.g. via successive halving). While effective, this procedure relies on training multiple models. More promisingly, Kan (2022) applies hypergradient descent to some hyperparameters of FedOpt. However, this work only considers certain client hyperparameters (notably, only hyperparameters of ClientUpdate), and relies on hand-derived gradients (which also use various approximations).

Federated AD gives us a straightforward way to compute $\partial L/\partial\alpha$ for hyperparameters without the need to derive explicit derivative formulas. In order to apply federated AD, we augment each round of FedOpt (Algorithm 1) with a computation in which 1) the server broadcasts its updated model to a set of clients, 2) each client computes their associated loss, and 3) the server receives the average of client loss. By using federated AD to differentiate through this, we can arrive at the desired derivative. This can be done via any of the modes discussed in Section 3.

Fig. 4 gives the associated federated computational graph to which we apply federated AD, specialized to a server-side hyperparameter. This federated computational graph has two components: A single round of FedOpt (essentially the same graph as Fig. 3), followed

Algorithm 2 FedOpt with hypergradient descent.

Input: x_0`@SERVER`, α_0, η`@SERVER`, $\{z_i\}$`@CLIENTS`
for $t = 0, \cdots, T-1$ **do**
    Sample a cohort C_t of participating clients.
    Server receives $x_{t+1} = \textsc{FedOptRound}(x_t, \alpha_t, \{z_i\}_{i \in C_t})$ and $\partial f(x_{t+1})/\partial \alpha_t$ via federated AD, as in Fig. 4.
    Server computes $\alpha_{t+1} = \alpha_t - \eta(\partial f(x_{t+1})/\partial \alpha_t)$.
end for

by a federated loss computation. This produces some loss measurement $L(x)$. By applying federated AD (forward-mode, reverse-mode, or mixed-mode), we can differentiate through Fig. 4 to get $\partial L/\partial \alpha$. Once we have this derivative, we can then perform a gradient-descent step to update α. High-level pseudo-code for this entire procedure is given in Algorithm 2. To simplify notation, we let $\textsc{FedOptRound}(x, \alpha, \{z_i\}_{i \in C})$ denote the model update produced by a single round of FedOpt, using some hyperparameter α, with client data $\{z_i\}_{i \in C}$ for a set C of participating clients. Note that other first-order optimization methods could be used to update the hyperparameter α.

We make a few important observations about Fig. 4 and federated AD.

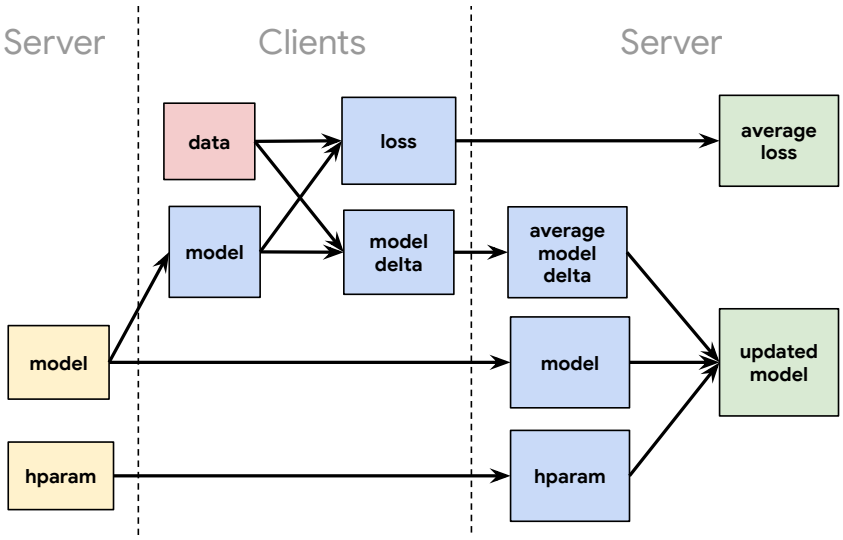
1. The augmented loss computation only provides a stochastic estimate of the true loss $f(x)$, and therefore a stochastic hypergradient $\partial f(x)/\partial \alpha$. Generally, this will depend on which clients participate during the augmented loss computation. However, the set of clients which participate in the augmented loss computation need not be the same as the clients used for the FedOpt model update. This is useful in FL settings where clients have limited availability, and may drop out between computations.

2. Due to the serial nature of the FedOpt round and augmented loss computation, we effectively double the number of communication rounds needed (relative to FedOpt). This can be important in FL settings (especially cross-device settings) where synchronization costs are high (Bonawitz et al., 2019). In order to avoid this, we can parallelize the FedOpt computation and augmented loss computation into a single communication round, as pictured in Fig. 5.

5. Learned Server Optimization

We now present an empirical exploration of federated hypergradient descent as applied to server optimization hyperparameters. We instantiate the FedOpt framework with the FedAvgM algorithm (Hsu et al., 2019): namely, Algorithm 1 where ClientUpdate is E epochs of SGD (as in FedAvg), and where ServerUpdate is a single step of SGD with momentum. We differentiate with respect to learning rate and momentum as detailed in Section 4 to pursue twin questions. First: can following gradients make FedAvgM easier to tune? Second: can following gradients recover and surpass performance of fixed hyperparameter settings for FedAvgM?

Figure 5: Federated computational graph used to compute hypergradients of server hyperparameters in FedOpt. All server→client communication is done via `federated_broadcast`, while all client→server is done via `federated_mean`. This computation produces some updated model and an estimate of the loss of that model. In contrast to Fig. 4, model training and loss computation (for the purposes of computing hypergradients) are done in parallel, potentially across different sets of clients. Note that applying federated AD to compute the derivative of the average loss with respect to the hyperparameter requires chaining two of these graphs together (in order to create a path from "hparam" to "average loss"), though this can be performed by the server after the fact leveraging only a direct application of federated AD to the graph presented here.



5.1 Experimental Setup

We tested our algorithm across five tasks adapted from Reddi et al. (2021). We used four data sets: CIFAR-100 (Krizhevsky, 2009), EMNIST (Cohen et al., 2017), Shakespeare (McMahan et al., 2017), and Stack Overflow (Authors, 2019). For EMNIST, we train a relatively small CNN for character recognition. For CIFAR-100, we train a modified ResNet-18 model, replacing BatchNorm layers with GroupNorm (Hsieh et al., 2020). For Shakespeare, we trained an RNN for next-character-prediction. For Stack Overflow, we performed tag prediction using a logistic regression on bag-of-words vector (SO TP) and trained an RNN to do next-word-prediction (SO NWP).

Throughout, we compare the "accuracy" of the learned model on the test split of each of the data sets above. This refers to the standard multi-class prediction accuracy for the CIFAR-100, EMNIST, Shakespeare, and Stack Overflow NWP tasks. For Stack Overflow TP, the "accuracy" refers to recall@5, following the convention in Reddi et al. (2021).

We use $E = 1$ epochs of mini-batch SGD in ClientUpdate throughout, and use the same batch sizes for each task as in (Charles et al., 2021). We set the per-client weights p_k to the number of examples in each client's data set (example-weighting). We sample 50

clients uniformly at random at each communication round. We use the same number for any derivative computations performed via federated AD, though these need not be coupled.

We compare various initialization strategies for the client learning rate, server learning rate, and server momentum. In each case, we either set it to some default value, or a random value. The random initializations use the same sampling strategy across tasks, detailed in Appendix A. The default client learning rates are taken from the tuned learning rates in (Reddi et al., 2021). The default server learning rate is set to 1.0, and the default momentum value is set to 0.9. We then compare random/default client learning rates and random/default server parameters, for a total of four initialization strategies.

We compute derivatives of the loss with respect to the server learning rate and momentum using a limited implementation of mixed-mode federated AD from Section 3, as detailed in Appendix A. We use the "parallelized" approach to Algorithm 2 presented in Fig. 5. Crucially, we *do not tune the hypergradient descent optimizer* we used to adjust the server momentum and learning rate parameters over time. We performed some initial small tuning on the EMNIST task, and simply reused this setting for all the rest. Throughout, our hypergradient descent optimizer is SGD with a learning rate of 0.01.

5.2 Results

We perform 50 random trials for each of the 4 default/random combinations and task. Across all initializations, learned learning rate and momentum values essentially always outperformed their fixed counterparts on all metrics we evaluated. We consider one of our initialization settings to be particularly interesting, and we discuss these results in detail: initialization to some 'default' server optimizer settings (learning rate of 1.0 and momentum of 0.9, a standard baseline setup in FL), and random client learning rates. Note that we see qualitatively similar results for the other three combinations of default/random, see Appendix A.

The default server / random client setting is intended to replicate a 'first-pass' hyperparameter tuning attempt for a FL practitioner. Without adaptive optimizers, FL with server learning rate of 1.0 is usually a reasonable default; indeed, the original formulation of FedAvg assumes a server learning rate of 1.0 when translated to the bi-level optimization setting of Reddi et al. (2021). The momentum parameter of 0.9 served as a well-tuned default in Reddi et al. (2021), though usually some small improvement can be found for a given task by tuning the momentum value. Little guidance exists, however, for choosing a client learning rate, and the optimal client learning rate for a federated tasks often differs substantially from the optimal learning rate for the same task in the centralized setting. In practice, this is effectively always swept over. This client learning rate sweep comes at considerable computational cost, particularly from the overall system health perspective, where tasks may compete for resources at training time (Bonawitz et al., 2019).

As shown in Table 4, allowing automatic tuning of learning rates and momentum parameters from their default values *always* increases the maximum accuracy achieved in this sweep over client learning rates; again, entirely in the absence of tuning the optimizer used to compute the hyperparameter updates.

Another important aspect of learned algorithms is their ability to reduce total compute time by requiring fewer random trials. To estimate the effects of this, we use a bootstrap analysis: For each number of trials $n \leq 50$, we pick n out of the 50 trials with replacement,

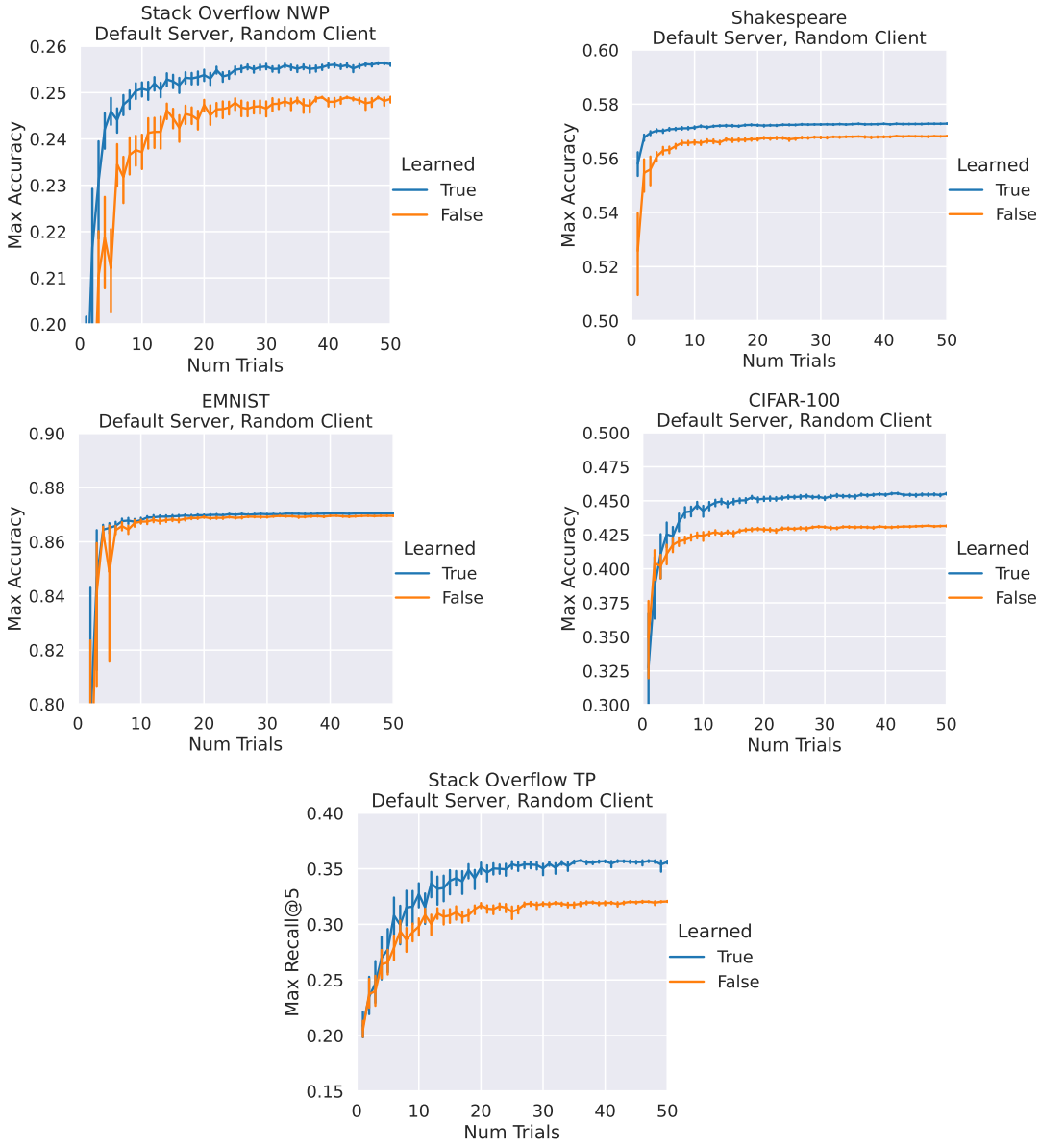


Figure 6: Estimated max accuracy from multiple trials of default server / random client initialization, as described above. 50 different seeds were used to determine the initializations, so that fixed and learned server optimizer hyperparameters share the same set of client learning rates and the same client sampling order. The plots were generated by computing a bootstrap estimate of the maximum test accuracy across $n \leq 50$ trials.

Learned	SO NWP	Shakespeare	EMNIST	CIFAR100	SO Tag
True	**25.7**	**57.3**	**87.1**	**45.6**	**35.8**
False	25.0	56.8	87.0	43.2	32.2

Table 4: Maximum test accuracy (%) across 50 trials when using federated AD to learn the learning rate and momentum parameters of server-side SGD. We initialize with learning rate 1.0 and momentum 0.9, and a random client learning rate.

and compute the maximum test accuracy across the n trials. We do this repeatedly to produce mean and variance estimates of the maximum test accuracy attained by running $n \leq 50$ trials. We plot the results for each task in Fig. 6. We see that when running the same experimental setup with fewer samples, we can attain computational savings by doing gradient-based adjustment of the server optimizer parameters. Generally, one may expect to replicate the max accuracy from the unlearned settings with one-tenth the trials.

The trajectories of the learned learning rate and momentum parameters also yield insight into the dynamics of FL algorithms. In Fig. 7, we plot the trajectories of server learning rate and momentum parameters for Shakespeare and SO NWP trials corresponding to the optimal setting of client learning rate over our random sweep. These trajectories reflect distinct features of the tasks under consideration: SO NWP initially *decreases* from its learning rate of 1.0 and momentum of 0.9, before increasing back up to slightly above these values, then slowly decaying. This initial decrease has a profound effect on training dynamics. Our default server / default client experiments, which we initialized with the optimal values of the sweep used by Reddi et al. (2021), were significantly more stable in the learned-hyperparameter setting: 23/50 of the non-learned experiment runs diverged very early on in training, but *none* of our learned experiments diverged. Meanwhile, the Shakespeare task proved itself difficult to tune with fixed hyperparameters; low server LR values eventually outperform large values in validation accuracy, seemingly due to an analog of over-fitting, with the appropriate hyperparameter choice therefore being highly dependent on the number of rounds for which one runs one's algorithm. Fig. 7 demonstrates a smooth transition from higher to lower learning rates, which enables a single learned Shakespeare task to effectively match the performance curve of multiple fixed hyperparameter specifications.

6. Learned Client Weighting

The parameters learned in Section 5 only affected server-side computation; federated AD, however, can differentiate with respect to parameters anywhere in an FL system. We now discuss how to use federated AD to learn the client weights ρ_i in Algorithm 1. While these weights are fixed in Algorithm 2, we apply federated hypergradient descent by making them a function of some hyperparameter. In order to subsume both example- and uniform-weighting, we let $\rho_i = n_i^q$ where q is a smooth function of a learnable hyperparameter γ (see Appendix B for details). Obviously, $q = 0$ recovers uniform weighting and $q = 1$ recovers example weighting. We then apply Algorithm 2 to learn γ in tandem with the model.

We broadcast γ to the clients at each round, who use it to compute their local weight in Algorithm 1. We then perform the same procedure as Algorithm 2, where we use mixed-mode

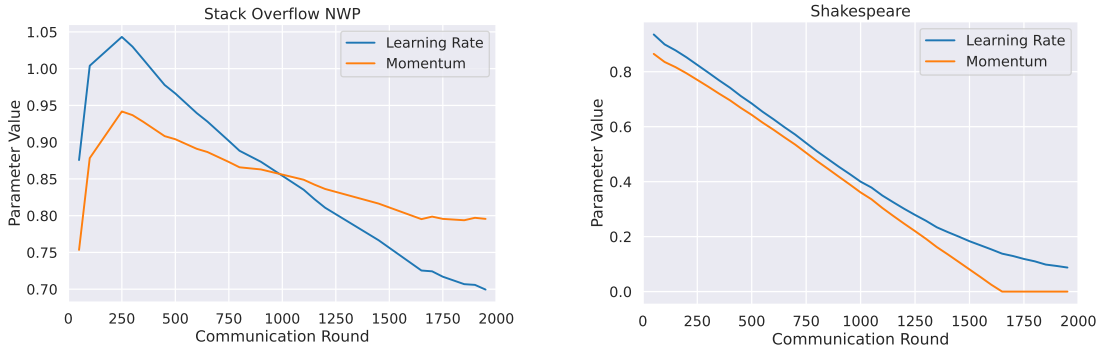


Figure 7: Example learned server learning rate and momentum trajectories for the Stack Overflow NWP and Shakespeare tasks.

federated AD to compute $\partial L/\partial\gamma$, where L is the empirical risk function. The corresponding federated computational graph is similar to Fig. 4, except that the parameter γ governing the weight exponent is broadcast to clients. The remainder of the graph is identical. We give a full picture in Fig. 12. This computation can also be parallelized, as in Fig. 5.

6.1 Experimental Setup

Now that we can apply federated AD, we use Algorithm 2 to learn the model via FedOpt in tandem with the client weight parameter γ via hypergradient descent. We apply Algorithm 2 to the same benchmark tasks in Section 5, with a few minor changes. First, we special case FedOpt to FedAvg by setting ServerUpdate to SGD with learning rate of 1.0. The client learning rate is set to the tuned defaults discussed in Section 5. Last, we use Adam (Kingma and Ba, 2015) in our hypergradient descent step (rather than SGD), with learning rate 0.01. All other implementation details are the same.

On all 5 tasks (CIFAR-100, EMNIST, Shakespeare, SO NWP, and SO TP) we see no statistically significant difference between learned client weighting and example-weighting. For some, like the CIFAR-100 task, this is because all clients have the same number of examples. For others, the root cause of this similarity is unclear, though a large sweep over fixed parameter values for γ also demonstrated little difference between these tasks. We therefore conjecture that these tasks are insufficiently heterogeneous to provide an effective illustration of applying federated AD to learn q.

In order to investigate federated AD for this scenario, then, we use the synthetic logistic regression problem proposed by Li et al. (2020), Synthetic(α, β). This task has multiple forms of heterogeneity (measured by the parameters α and β) in addition to having unbalanced clients. Following the implementation used in (Li et al., 2020), the distribution of number of examples across clients follows a shifted log-normal distribution. For more details on this task, see Appendix B. In short, the task provides a setting where unweighted and client-weighted means perform quite differently (as noted by Li et al. (2020)), making it a useful candidate task for learned client weighting. We set $\alpha = 1, \beta = 1$ for our experiments, as this is the "most heterogeneous" form of the task investigated by Li et al. (2020) and Wang et al. (2020).

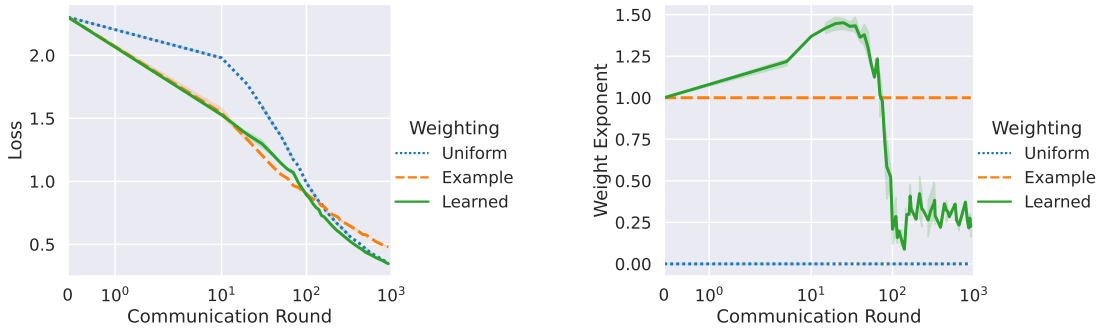


Figure 8: Loss and weighting exponent for the Synthetic$(1, 1)$ task, where we compare example-weighting, uniform weighting, and learned weighting.

6.2 Results

We present our results in Fig. 8. Several interesting findings are worth highlighting. First, while we initialize such that the learned weighting exponent starts at $q = 1$, we see comparable (though slightly worse) results when initializing at $q = 0$ (see Fig. 13). Second, in the fixed parameter setting, example-weighted averaging begins the training procedure by significantly outperforming uniform averaging. This phenomenon may be understood by noting that example-weighted means give higher weight to clients which took the most steps, and are therefore likely to have traveled furthest; updates therefore tend to be *larger* early on in training for example-weighted FedAvg. However, the heterogeneity of this task leads to the eventual reversal of performance, with uniform weighting outperforming example weighting by the end of training.

Notably, by using federated AD to learn the weight exponent, we can attain performance comparable to the *lower envelope* of these two curves. In other words, federated AD provides an automatic manner of interpolating between these two regimes in which different weighting is called for. This capacity for dynamic interpolation provides one potential solution to the thorny trade-offs between speed and accuracy in FL, studied in depth in (Charles and Konečný, 2021).

7. Conclusion

In this work, we describe a programmatic framework, federated automatic differentiation, for computing derivatives of a broad class of federated computations that are compatible with privacy-preserving technologies. Moreover, we showed that our framework yields a federated computation in the same class, so that we can be assured that derivatives can always be computed with formal privacy mechanisms as well. We presented multiple modes for federated AD and discussed their implications for target systems. We applied federated AD to federated hypergradient descent problems, and showed that it can improve accuracy and make finding good hyperparameters much less expensive.

This work is intended to be a starting point for the study of applying AD to FL. Some important open problems include developing software libraries capable of implementing

federated AD[9], developing improved FL algorithms through the use of federated AD, and thoroughly studying the behavior (theoretical and empirical) of "dynamic" FL algorithms that adjust their hyperparameters and operation in tandem with training.

Acknowledgments

We thank Brendan McMahan for comments on the manuscript; Roy Frostig for discussions on JAX's AD implementation; Wenjun Hu for thoughtful comments on distributed systems; Jascha Sohl-Dickstein for a dicussion of learned optimizers and evolution-strategies-based gradient computation; Matthew Streeter for early discussions on learned optimizers; Blaise Aguera y Arcas and Krzysztof Ostrowski for early support in conceptualizing FL as functional; and Jared Lichtarge and Shankar Kumar for helpful discussions of applications of federated AD.

Appendix A. Learned Server Optimization - Details

This section contains various details of the empirical evaluation in Section 5. In particular, we cover the implementation details, hyperparameter initialization schemes, and methods for computing gradients that are necessary to recreate our empirical results.

A.1 Hyperparameter Initialization

As discussed in Section 5, we initialized server hyperparameters (learning rate and momentum) and client hyperparameters (learning rate) either randomly, or with default settings. For random initialization, learning rates were chosen via a log-uniform distribution from the range $(10^{-3}, 10)$. Momentum values were chosen uniformly from the range $(0, 1)$. For default server optimizer settings, we used a learning rate of 1.0 and a momentum value of 0.9. For default client learning rate settings, we used the optimal values reported from the sweep in (Reddi et al., 2021). All random-setting experiments were repeated 50 times.

A.2 Implementation

The implementation which backed our learned server optimization experiments was parameterized by function and Jacobian computations, relying on hand-implementations of a few limited federated AD components to compose these together and compute derivatives of model losses with respect to server optimizer hyperparameters. We instantiated these parameterizations with hand-differentiated pairs of functions representing server optimizer updates implemented in TensorFlow. The federated portions of our programs could be implemented in terms of existing symbols in TensorFlow-Federated, particularly those defining functions for computing FedAvg and FedSGD. These computations were invoked in parallel, as represented in Fig. 5.

A subtle point of distinction may be made with respect to what precise relationship exists between the clients which feed the FedAvg computation and those which feed the FedSGD

9. Since the initial version of this work, we have developed a high-performance implementation of federated AD in JAX; see (Rush et al., 2024).

procedure. At least three options come to mind immediately: using identical clients; using clients sampled from the same set ('population'), though using two different samples for each invocation of the two computations; finally, using clients sampled from two entirely different populations.

We consider the final two to be more compatible with current cross-device FL systems, where sampling a fresh batch of clients and asking them for the relatively lightweight computation of a single gradient is significantly cheaper than either requiring persistent client-server connections or increasing the workloads and payloads of individual clients. We generally expect any differences between leveraging a fresh sample from the same population and a sample from a completely different population to be artifacts of the task under consideration; for this reason, we focused on sampling clients for both purposes from a single population. Such 'gradients from train clients' experiments populate all the figures used here. On some tasks, however, notably Shakespeare, leveraging gradients from validation clients demonstrated a remarkable ability to avoid the problem of over-fitting to the *training population* which seems to be endemic to federated Shakespeare training.

A.3 Additional Experimental Results

Here we provide results on the other experiments run in support of Section 5, where we only presented the results for random client / 'default' server hyperparameter initializations.

Below we present the results for the other three combinations (random server/random client, random server/default client, and default server/default client). As in Section 5, we run 50 trials for each (across five tasks), and use a bootstrap analysis to estimate the maximum test accuracy when running $n \leq 50$ trials. The results are given in Figures 9, 10, and 11. In nearly all cases, we see that the learned server optimizer can attain a higher maximum accuracy in many fewer trials.

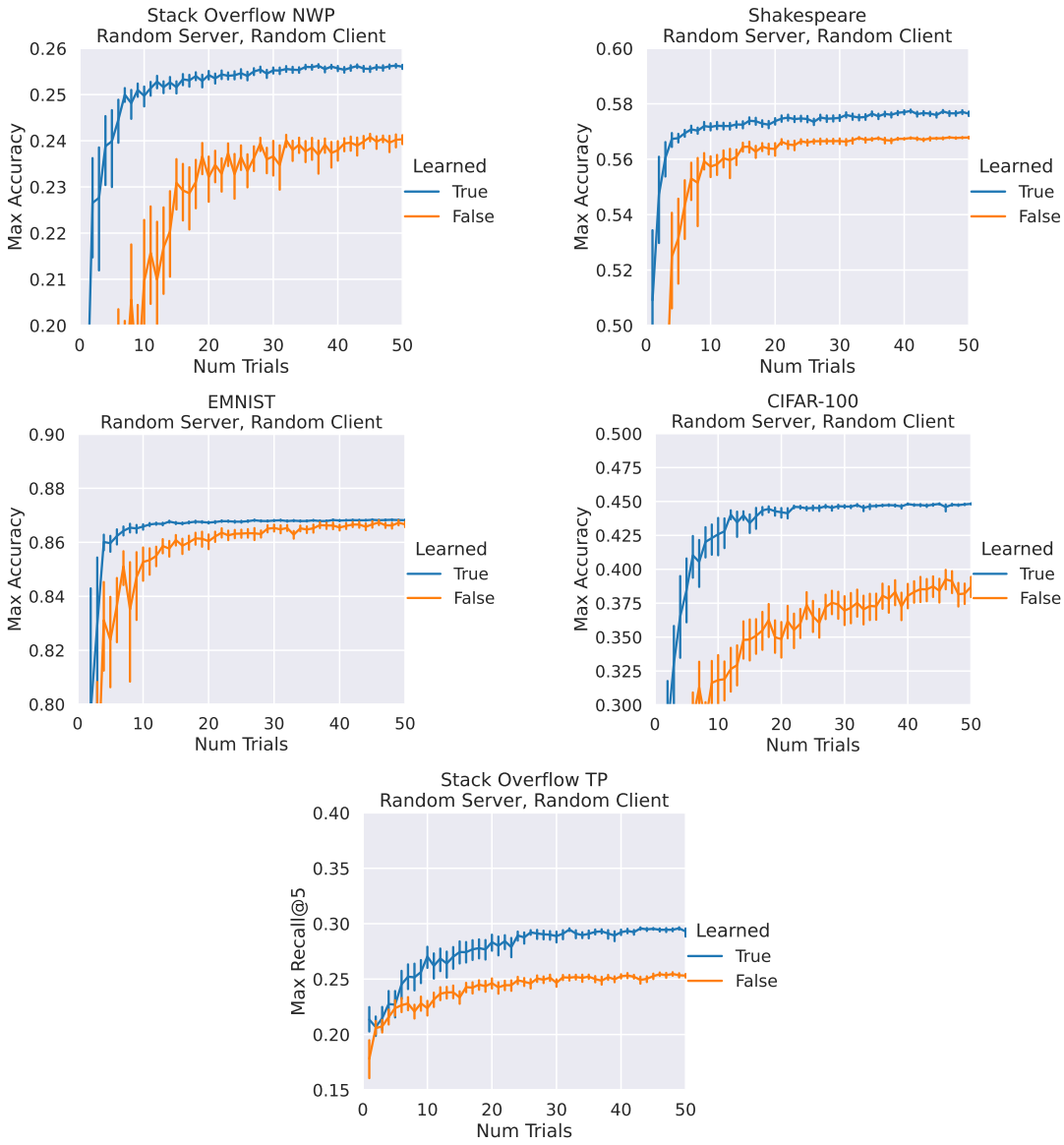


Figure 9: Estimated max accuracy from multiple trials of random server / random client initialization, as described in Section 5. 50 different seeds were used to determine the initializations, so that fixed and learned server optimizer hyperparameters share the same initial parameters and client sampling order. The plots were generated by computing a bootstrap estimate of the maximum test accuracy across $n \leq 50$ trials.

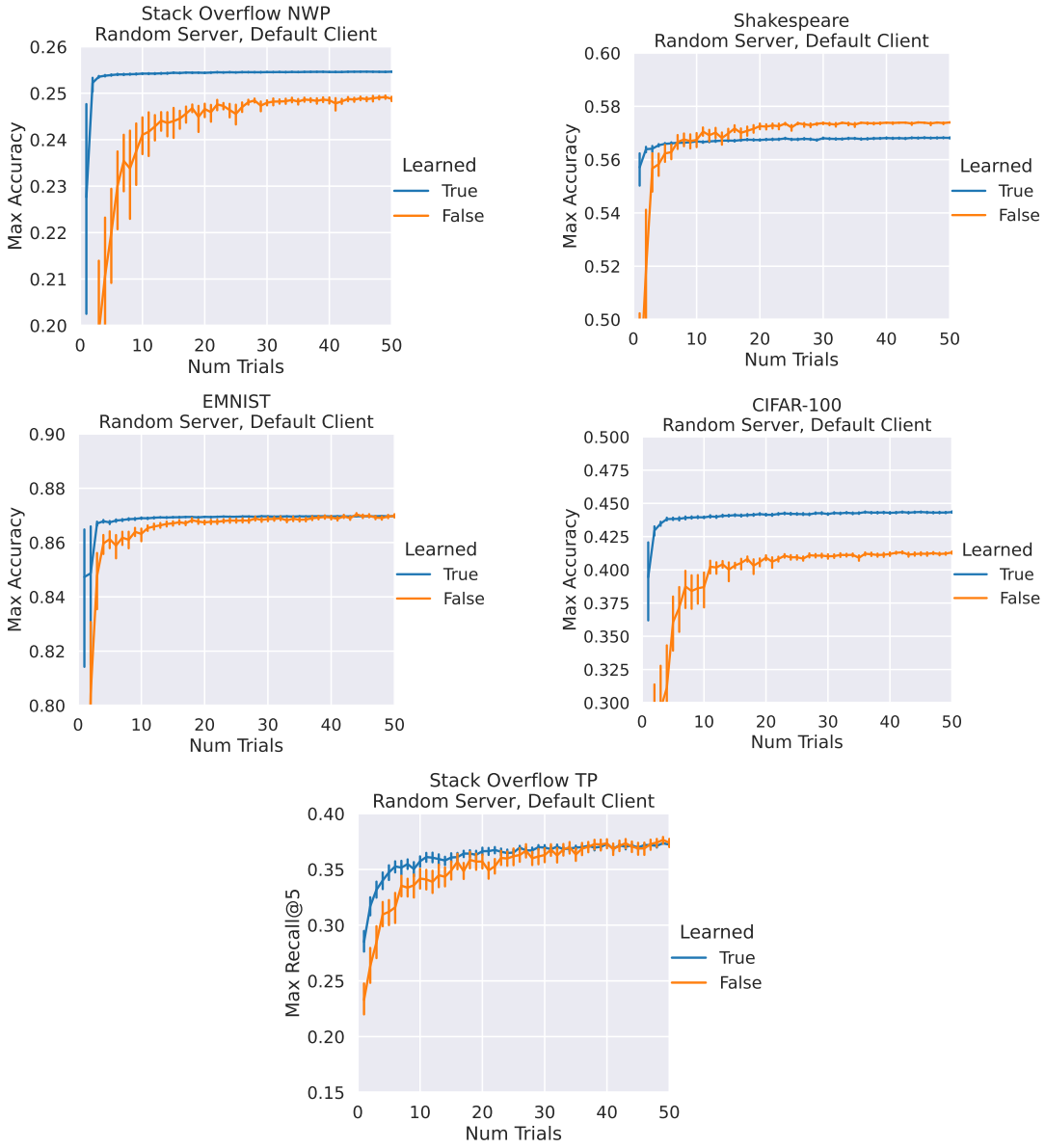


Figure 10: Estimated max accuracy from multiple trials of random server / default client initialization, as described in Section 5.

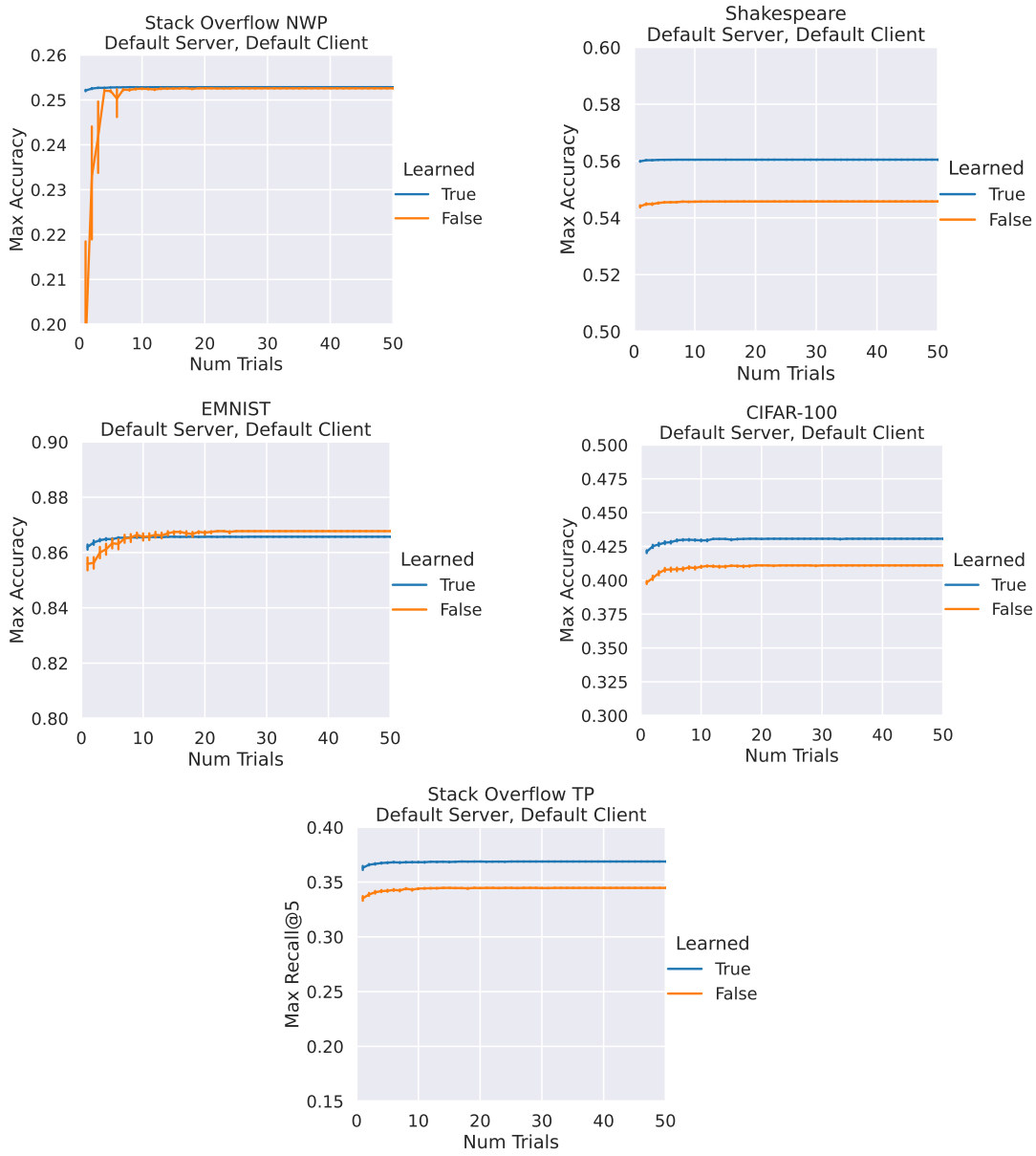
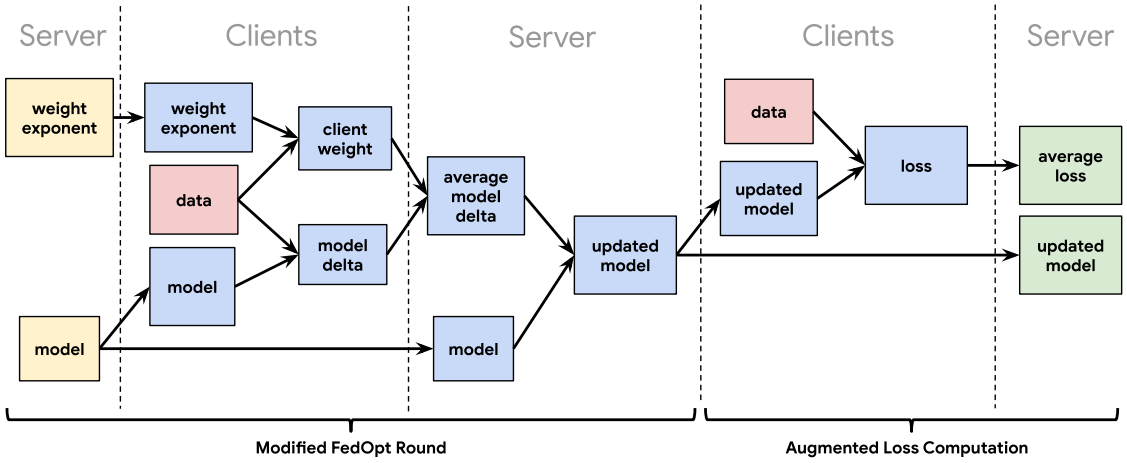


Figure 11: Estimated max accuracy from multiple trials of default server / default client initialization, as described in Section 5.

Figure 12: Federated computational graph used to compute hypergradients of the client weight exponent used in Section 6. All server→client communication is done via `federated_broadcast`, while all client→server is done via `federated_mean`. This computation produces some updated model and an estimate of the loss of that model. By applying federated AD, we can compute the desired hypergradients.



Appendix B. Learned Client Weighting - Details

In this section, we discuss details concerning the experimental investigations in Section 6. In particular, we provide a complete description of the computational graph to which we apply federated AD, details on the synthetic data set used, and additional experimental results.

B.1 Federated Computational Graph

Here we provide a computational graph demonstrating the parameterization of client weighting as discussed in Section 6.The approach largely mirrors the approach used to apply federated AD to server optimizer hyperparameters in Fig. 4. The key difference is that the client weight parameter must be broadcast to the clients at each round, since the server updates it between rounds. The resulting procedure is given in Fig. 12. Note that this can be parallelized in an almost identical way to that of Fig. 5.

B.2 Synthetic Logistic Regression Task

We use the $Synthetic(\alpha, \beta)$ task proposed by Li et al. (2020). Each client i has the following parameters sampled independently:

- A number of example $n_i \sim \mathcal{P}$.
- A model parameter $u_i \sim \mathcal{N}(0, \alpha)$.
- A mean vector $v_i \in \mathbb{R}^{60}$ with elements drawn from $\mathcal{N}(\beta_i', 1)$ where $\beta_i' \sim \mathcal{N}(0, \beta)$.

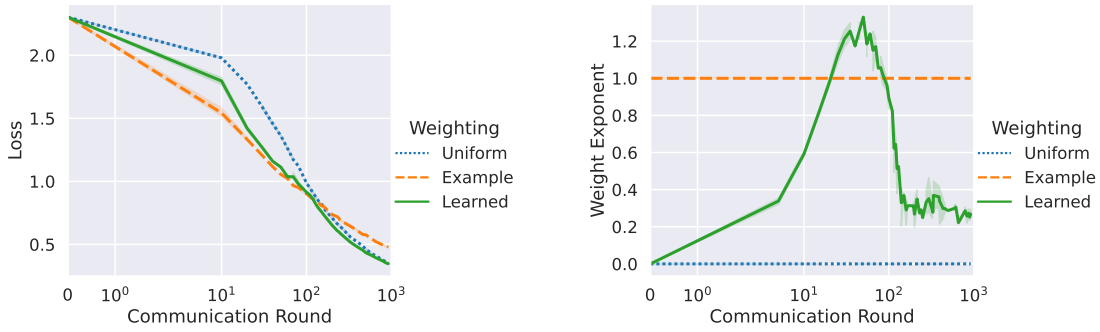


Figure 13: Loss and weighting exponent for the Synthetic$(1, 1)$ task, where we compare example-weighting, uniform weighting, and learned weighting. While we initialize the learned weighting exponent at $q = 0$, we see comparable (though slightly better) results when initializing at $q = 1$ (see Fig. 8).

Each client i has a "ground truth" multi-class logistic regression model $f(x) = \text{argmax}(\sigma(W_i x + b_i))$ where σ is the softmax function, $W_i \in \mathbb{R}^{10\times 60}$, and $b_i \in \mathbb{R}^{10}$. The entries of W_i and b_i are sampled independently from $\mathcal{N}(u_i, 1)$. Each client generates data $\{(x_{i,n}, y_{i,n})\}_{n=1}^{n_i}$ via $x_{i,n} \sim \mathcal{N}(v_i, \Sigma)$, $y_{i,n} = f(x_{i,n})$. Here, Σ is a diagonal covariance matrix with $\Sigma_{j,j} = j^{-1.2}$.

The remaining detail is the distribution $\mathcal{P}$ for the number of examples per client. We use the same implementation from Li et al. (2020), where $\mathcal{P}$ is a shifted log-normal distribution. Specifically, $n_i = 50 + \lfloor \exp(m_i) \rfloor$ where $m_i \sim \mathcal{N}(4, 2)$. In our empirical evaluation in Section 6, we let there be 100 clients overall, and sample 50 of them at each round.

B.3 Additional Experimental Results

As in Section 6, we compare uniform weighting, example weighting, and learned weighting variants of the FedOpt algorithm (Algorithm 1) on the synthetic logistic regression task detailed above. While Fig. 8 compared the three in a regime where the learned weight starts off at example weighting, we compare the three when the learned weight starts off at uniform weighting in Fig. 13.

We see that the learned weighting does better than the uniform initially (though there is some gap with example weighting) but eventually does better than both (and significantly better than example weighting, which levels off). Looking at the learned weight exponent, it quickly increases initially (in fact, to a weighting scheme where clients with more examples are weighted more heavily) and then goes back to some thing closer to uniform weighting eventually. In short, the learned weighting seems to approximate the lower envelope of the loss for uniform and example weighting.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: a

system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 3988–3996, Red Hook, NY, USA, 2016. Curran Associates Inc. ISBN 9781510838819.

The TensorFlow Federated Authors. TensorFlow Federated Stack Overflow dataset, 2019. URL `https://www.tensorflow.org/federated/api_docs/python/tff/simulation/datasets/stackoverflow/load_data`.

Friedrich L Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

Atilim Gunes Baydin, Robert Cornish, David Martínez-Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018a. URL `https://openreview.net/forum?id=BkrsAzWAb`.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018b. URL `http://jmlr.org/papers/v18/17-468.html`.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural optimizer search with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 459–468. PMLR, 2017. URL `http://proceedings.mlr.press/v70/bello17a.html`.

Y Bengio, S Bengio, and J Cloutier. Learning a synaptic learning rule. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 969–vol. IEEE, 1991.

Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12 (8):1889–1900, 2000. doi: 10.1162/089976600300015187.

K.A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1175–1191, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3133982. URL `https://doi.org/10.1145/3133956.3133982`.

Kallista Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan

McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 374–388, 2019. URL https://proceedings.mlsys.org/paper/2019/file/bd686fd640be98efaae0091fa301e613-Paper.pdf.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. JAX: composable transformations of Python+ NumPy programs. *Version 0.2*, 5:14–24, 2018.

Zachary Charles and Jakub Konečný. Convergence and accuracy trade-offs in federated learning and meta-learning. In Arindam Banerjee and Kenji Fukumizu, editors, *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 2575–2583. PMLR, 13–15 Apr 2021. URL https://proceedings.mlr.press/v130/charles21a.html.

Zachary Charles and Keith Rush. Iterated vector fields and conservatism, with applications to federated learning. In Sanjoy Dasgupta and Nika Haghtalab, editors, *Proceedings of The 33rd International Conference on Algorithmic Learning Theory*, volume 167 of *Proceedings of Machine Learning Research*, pages 130–147. PMLR, 29 Mar–01 Apr 2022. URL https://proceedings.mlr.press/v167/charles22a.html.

Zachary Charles, Zachary Garrett, Zhouyuan Huo, Sergei Shmulyian, and Virginia Smith. On large-cohort training for federated learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 20461–20475. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/ab9ebd57177b5106ad7879f0896685d4-Paper.pdf.

Zachary Charles, Kallista Bonawitz, Stanislav Chiknavaryan, Brendan McMahan, et al. Federated select: A primitive for communication-and memory-efficient federated learning. *arXiv preprint arXiv:2208.09432*, 2022.

Mingqing Chen, Rajiv Mathews, Tom Ouyang, and Françoise Beaufays. Federated learning of out-of-vocabulary words. *CoRR*, abs/1903.10635, 2019. URL http://arxiv.org/abs/1903.10635.

Bruce Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994.

Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. EMNIST: Extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20:55:1–55:21, 2019. URL http://jmlr.org/papers/v20/18-598.html.

Alireza Fallah, Aryan Mokhtari, and Asuman Ozdaglar. Personalized federated learning with theoretical guarantees: A model-agnostic meta-learning approach. *Advances in Neural Information Processing Systems*, 33:3557–3568, 2020.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

Andreas Griewank and Andrea Walther. *Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition*. SIAM, 2008. ISBN 978-0-89871-659-7. doi: 10.1137/1.9780898717761. URL https://doi.org/10.1137/1.9780898717761.

Mansura Habiba and Barak A Pearlmutter. Neural network based on automatic differentiation transformation of numeric iterate-to-fixedpoint. In *2021 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–6. IEEE, 2021.

Andrew Hard, Kanishka Rao, Rajiv Mathews, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *CoRR*, abs/1811.03604, 2018. URL http://arxiv.org/abs/1811.03604.

Andrew Hard, Kurt Partridge, Cameron Nguyen, Niranjan Subrahmanya, Aishanee Shah, Pai Zhu, Ignacio López-Moreno, and Rajiv Mathews. Training keyword spotting models on non-IID data with federated learning. In Helen Meng, Bo Xu, and Thomas Fang Zheng, editors, *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, pages 4343–4347. ISCA, 2020. doi: 10.21437/INTERSPEECH.2020-3023. URL https://doi.org/10.21437/Interspeech.2020-3023.

Charlie Hou, Kiran Koshy Thekumparampil, Giulia Fanti, and Sewoong Oh. FedChain: Chained algorithms for near-optimal communication cost in federated learning. In *International Conference on Learning Representations*, 2021.

Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip Gibbons. The non-IID data quagmire of decentralized machine learning. In *International Conference on Machine Learning*, pages 4387–4398. PMLR, 2020.

Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. Measuring the effects of non-identical data distribution for federated visual classification. *CoRR*, abs/1909.06335, 2019. URL http://arxiv.org/abs/1909.06335.

Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, Kaikai Wang, Anthony Shoumikhin, Jesik Min, and Mani Malek. PAPAYA: practical, private, and scalable federated learning. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022. URL https://proceedings.mlsys.org/paper/2022/hash/f340f1b1f65b6df5b5e3f94d95b11daf-Abstract.html.

Alex Ingerman and Krzysztof Ostrowski. Introducing TensorFlow Federated, Mar 2019. URL https://blog.tensorflow.org/2019/03/introducing-tensorflow-federated.html.

Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *Found. Trends Mach. Learn.*, 14(1-2):1–210, 2021. doi: 10.1561/2200000083. URL https://doi.org/10.1561/2200000083.

Andrew K. Kan. Federated hypergradient descent. *CoRR*, abs/2211.02106, 2022. doi: 10.48550/ARXIV.2211.02106. URL https://doi.org/10.48550/arXiv.2211.02106.

Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *International Conference on Machine Learning*, pages 5132–5143. PMLR, 2020.

Mikhail Khodak, Renbo Tu, Tian Li, Liam Li, Maria-Florina F Balcan, Virginia Smith, and Ameet Talwalkar. Federated hyperparameter tuning: Challenges, baselines, and connections to weight-sharing. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 19184–19197. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/a0205b87490c847182672e8d371e9948-Paper.pdf.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

Ke Li and Jitendra Malik. Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441*, 2017.

Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems*, 2:429–450, 2020.

Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. In *International Conference on Machine Learning*, pages 2247–2255. PMLR, 2017.

Grigory Malinovskiy, Dmitry Kovalev, Elnur Gasanov, Laurent Condat, and Peter Richtarik. From local SGD to local fixed-point methods for federated learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 6692–6701. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/malinovskiy20a.html.

Oleksandr Manzyuk, Barak A Pearlmutter, Alexey Andreyevich Radul, David R Rush, and Jeffrey Mark Siskind. Perturbation confusion in forward automatic differentiation of higher-order functions. *Journal of Functional Programming*, 29:e12, 2019.

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

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Meta-learning update rules for unsupervised representation learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019a. URL https://openreview.net/forum?id=HkNDsiC9KQ.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4556–4565. PMLR, 09–15 Jun 2019b. URL https://proceedings.mlr.press/v97/metz19a.html.

Luke Metz, James Harrison, C. Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, and Jascha Sohl-Dickstein. Velo: Training versatile learned optimizers by scaling up. *CoRR*, abs/2211.09760, 2022. doi: 10.48550/ARXIV.2211.09760. URL https://doi.org/10.48550/arXiv.2211.09760.

Konstantin Mishchenko, Grigory Malinovsky, Sebastian U. Stich, and Peter Richtárik. ProxSkip: Yes! local gradient steps provably lead to communication acceleration! finally! In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 15750–15769. PMLR, 2022. URL https://proceedings.mlr.press/v162/mishchenko22b.html.

Aritra Mitra, Rayana H. Jaafar, George J. Pappas, and Hamed Hassani. Linear convergence in federated learning: Tackling client heterogeneity and sparse gradients. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman

Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 14606–14619, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/7a6bda9ad6ffdac035c752743b7e9d0e-Abstract.html.

Uwe Naumann. Optimal jacobian accumulation is np-complete. *Math. Program.*, 112(2): 427–441, apr 2008. ISSN 0025-5610.

Deniz Oktay, Johannes Ballé, Saurabh Singh, and Abhinav Shrivastava. Scalable model compression by entropy penalized reparameterization. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=HkgxW0EYDS.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Reese Pathak and Martin J. Wainwright. Fedsplit: an algorithmic framework for fast federated optimization. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/4ebd440d99504722d80de606ea8507da-Abstract.html.

Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier C. van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, Sudeep Agarwal, Julien Freudiger, Andrew Byde, Abhishek Bhowmick, Gaurav Kapoor, Si Beaumont, Áine Cahill, Dominic Hughes, Omid Javidbakht, Fei Dong, Rehan Rishi, and Stanley Hung. Federated evaluation and tuning for on-device personalization: System design & applications. *CoRR*, abs/2102.08503, 2021. URL https://arxiv.org/abs/2102.08503.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4092–4101. PMLR, 2018. URL http://proceedings.mlr.press/v80/pham18a.html.

Krishna Pillutla, Sham M. Kakade, and Zaïd Harchaoui. Robust aggregation for federated learning. *IEEE Trans. Signal Process.*, 70:1142–1154, 2022. doi: 10.1109/TSP.2022.3153135. URL https://doi.org/10.1109/TSP.2022.3153135.

Swaroop Ramaswamy, Rajiv Mathews, Kanishka Rao, and Françoise Beaufays. Federated learning for emoji prediction in a mobile keyboard. *CoRR*, abs/1906.04329, 2019. URL http://arxiv.org/abs/1906.04329.

Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. Adaptive federated optimization.

In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=LkFG3lB13U5.

Keith Rush, Zachary Charles, and Zachary Garrett. Fax: Scalable and differentiable federated primitives in jax. *arXiv preprint arXiv:2403.07128*, 2024.

Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-learning with latent embedding optimization. 2019. URL https://openreview.net/forum?id=BJgklhAcK7.

Mark Sandler, Max Vladymyrov, Andrey Zhmoginov, Nolan Miller, Tom Madams, Andrew Jackson, and Blaise Agüera y Arcas. Meta-learning bidirectional update rules. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 9288–9300. PMLR, 2021. URL http://proceedings.mlr.press/v139/sandler21a.html.

J Schmidhuber. On learning how to learn learning strategies (technical report FKI-198-94). *Fakultat fur Informatik*, 1994.

Jeffrey Mark Siskind and Barak A Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode ad. 2005.

Davoud Ataee Tarzanagh, Mingchen Li, Christos Thrampoulidis, and Samet Oymak. FedNest: Federated bilevel, minimax, and compositional optimization. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 21146–21179. PMLR, 2022. URL https://proceedings.mlr.press/v162/tarzanagh22a.html.

Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10553–10563. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/vicol21a.html.

Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H Vincent Poor. Tackling the objective inconsistency problem in heterogeneous federated optimization. *Advances in neural information processing systems*, 33:7611–7623, 2020.

Ziyao Wang, Jianyu Wang, and Ang Li. FedHyper: A universal and robust learning rate scheduler for federated learning with hypergradient descent. In *The Twelfth International Conference on Learning Representations*, 2023.

Olga Wichrowska, Niru Maheswaranathan, Matthew W. Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine*

Learning Research, pages 3751–3760. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/wichrowska17a.html.

Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving google keyboard query suggestions. *CoRR*, abs/1812.02903, 2018. URL http://arxiv.org/abs/1812.02903.

Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *International Conference on Machine Learning*, pages 5650–5659. PMLR, 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=r1Ue8Hcxg.