Scatterbrained: A flexible and expandable pattern for decentralized machine learning

Miller Wilt[a], **Jordan K. Matelsky**[a,b], **and Andrew S. Gearhart**[a]

[a]Johns Hopkins Applied Physics Laboratory, Laurel, Maryland, United States
[b]University of Pennsylvania Department of Bioengineering, Philadelphia, Pennsylvania, United States

Federated machine learning is a technique for training a model across multiple devices without exchanging data between them. Because data remains local to each compute node, federated learning is well-suited for use-cases in fields where data is carefully controlled, such as medicine, or in domains with bandwidth constraints. One weakness of this approach is that most federated learning tools rely upon a central server to perform workload delegation and to produce a single shared model. Here, we suggest a flexible framework for decentralizing the federated learning pattern, and provide an open-source, reference implementation compatible with PyTorch.
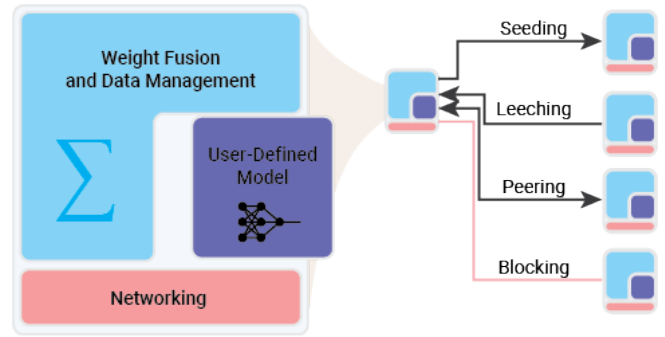
federated learning | distributed computing | edge computing | decentralized
Correspondence: *jordan.matelsky@jhuapl.edu*

Fig. 1. A proposed architecture for a decentralized machine learning node. At left, a single node is composed of: (1) a networking layer, to communicate with other compute nodes in the community, (2) a weight combination and data management component to choose parameter-sharing behavior and train a local model, and (3) a "pluggable" module in which a developer can add their machine learning model. At right, the node participates in four different types of network behavior: A "seeding" relationship in which the node sends but does not receive weights; a "leeching" relationship in which the node receives but does not send weights; a "peering" relationship in which the node both sends as well as receives; and a "blocking" relationship in which the node performs no communication with other nodes.

Introduction

Conventional machine learning technologies have achieved wide adoption due to their high accuracy and adaptability to diverse use-cases. To improve efficiency and minimize cost, data and machine learning tools are most commonly collocated, and neural networks are trained on large, centralized compute resources. This approach is popular, but is poorly suited to certain types of tasks, such as when a dataset contains sensitive data that cannot be shared due to logistical or legal restrictions (e.g., medical data), or in cases where it is impractical to transmit data to centralized resources (e.g., in large-scale edge computing applications, or in bandwidth-austere settings).

To address this problem, researchers have proposed *federated learning* (FL), a paradigm in which individual compute nodes train on their data locally, rather than sending data to a centralized server (1). Neural network parameters (or weights) — rather than raw data — are then shared with a central server, which returns a combined model to each of the edge nodes. In this way, all nodes benefit from their peers' training, without direct access to their peers' data. Such a method is most famously used by the Google GBoard predictive text feature on mobile phone keyboards (2).

One major limitation of the traditional federated learning approach is that it requires continuous and secure access to a trusted central server (3). Many implementations, such as TensorFlow Federated (4) or PySyft (3), make similar assumptions about node reliability, and leave it to the developer to resolve server trust as well as node dropout or drop-in connectivity behavior. Consistent communication with a remote compute node can not be guaranteed in all federated learning circumstances, and is almost certainly not the case in a decentralized FL environment, where the number of connections between nodes scales rapidly with the size of the cluster (5–8).

It is also critical that the cooperative learning algorithm be entirely decoupled from the machine learning model in question. In other words, the representation of the decentralized compute graph should have no bearing upon the model itself, and any machine learning model should be compatible with a decentralized learning framework. We enforce this requirement because we assert that adding a decentralized federated learning regime to an existing tool should not meaningfully impact the model or presentation of data. We illustrate the main differences between traditional machine learning, federated learning, and decentralized federated learning in **Fig. 3**.

In this work, we suggest critical features for a successful decentralized federated learning platform and share our reference implementation, *Scatterbrained*. Our framework satisfies the requirements of (1) allowing a developer to define use-case specific compute graph topologies, (2) enabling population- or node-level parameter-sharing behavior, (3) permitting node dropout or lossiness, and (4) decoupling machine learning from network communication (**Fig. 1**).
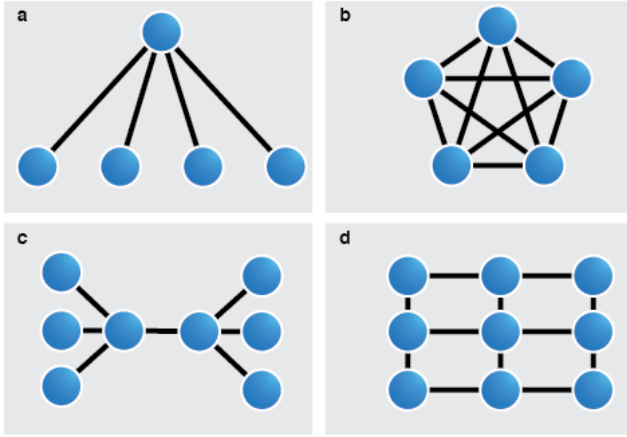
Fig. 2. Examples of federated learning compute-graph topology. We propose that a federated learning framework must support diverse compute-graph topologies. **a. Traditional federated learning.** An example of a centralized federated learning approach, with a central node responsible for data aggregation. **b. Fully-connected graph.** A fully connected graph enables each node to communicate with every other node, increasing bandwidth usage but maximizing the speed at which new information traverses the graph. **c. Local neighborhood graph.** Such a graph might be useful when many edge devices connect to a smaller number of higher-performance servers, where bandwidth and compute is inexpensive. **d. Grid layout.** A grid layout may be useful for geographically distributed nodes in order to improve bandwidth to peers while reducing construction cost.

Desired Traits of a Decentralized Learning Framework

Existing implementations of decentralized federated learning. There are several competing proposals of decentralized federated learning algorithms. (7–10) Most notably, we call attention to *Lalitha et al.* (7) and the *BrainTorrent* algorithm (8), two distinct approaches to cooperative learning. Importantly, these existing implementations are purpose-built, and do not afford the end-user sufficient flexibility to manipulate compute-graph topology or node-level data-sharing preferences. It is our belief that these properties should be developer-facing parameters, as different use-cases may call for different graph topology or data sharing strategies. That is, a universal federated learning framework should support all federated learning algorithms discussed in this manuscript, *including* centralized approaches.

Critical features for a federated learning framework. We propose that there is a set of necessary, adjustable parameters that must be present for a federated learning framework to be useful to the general machine learning community:

Compute-graph topology. The compute graph topology describes how nodes in a federated learning cluster connect to one another. In one extreme, nodes can connect to only a single (central) node: This describes a traditional, centralized federated learning algorithm (**Fig. 2a**). In the other extreme, each node may connect to every other node in the community, forming a fully connected graph (**Fig. 2b**). Depending on the use case, it may also be useful for clusters to adopt other connectivity patterns (**Fig. 2c,d**).

Parameter sharing preferences. Node-level parameter-sharing preferences refer to how individual nodes participate in cooperative learning. Each node may act as a *read-only*, *write-only*, or *read-and-write* resource in the graph. It is also useful to allow a node's behavior to vary over time based upon internal or external stimuli. For example, a node that has not encountered local data may start in *read-only* mode, accepting inputs from peers until it reaches a threshold volume of useful local data. After this node has its own useful new information to contribute to the network, it may switch into *read-and-write* mode. Other use-cases may necessitate other state transitions between these strategies.

Ephemeral nodes. In distributed edge applications, node network connectivity may be limited, leading to inconsistent node connectivity or permanent dropout. This effect is compounded when the size of the cluster approaches hundreds or thousands of compute nodes. To support applications where individual node network connectivity may be limited, a federated cluster must be robust to node dropout and intermittent bandwidth constraints. This is one of the primary components missing from most existing implementations of decentralized federated learning, and it is one that is most important to guarantee the resiliency of the network as a whole.

This is particularly important for Internet-of-Things devices (IoT), where network access is not only not guaranteed, but is sometimes deliberately withheld to improve energy efficiency (6).

Abstraction of the model from federation. We assert that the separation of the machine learning model from the federated learning business logic is a requirement for a successful decentralized federated learning framework. This is because tight integration of a model with the framework imparts an engineering burden on the developer–likely unacceptable for most machine learning applications. Further, by abstracting the machine learning model from federation logic, we enable use of a diverse set of learning tools, such as different ML libraries or different strategies for combining node-level models.

Reference Implementation. We have developed a Python library, *Scatterbrained*, which abstracts many of the implementation details of networking while enabling a developer to leverage the flexibility of the decentralized learning paradigm. In this work, we demonstrate Scatterbrained compatibility with the popular PyTorch (11) library, but we believe it is a straightforward extension to extend this work for compatibility with other machine learning libraries.

To maximize the usability of the Scatterbrained library for existing use-cases, we developed a simple API which abstracts low-level node communication across the compute graph and enables fine-grained customization of a node's parameter-sharing behavior. To perform low-overhead communication between Scatterbrained nodes and provide a developer with the ability to design a custom compute graph, we built a communication layer utilizing the 0MQ library (12).

This high-level networking API performs communication on behalf of the model training and parameter-pooling modules. Our intention is for all *Scatterbrained* operation to be largely
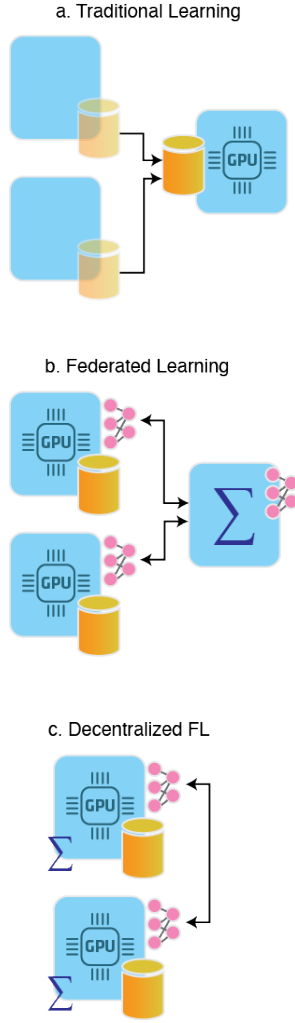
Fig. 3. a. Traditional learning. Edge nodes upload raw data to a central server. The central server performs all computation. **b. Federated learning.** Edge nodes perform local computation and training. A central server performs parameter pooling. **c. Decentralized federated learning.** Edge nodes perform local computation and training. Each node performs pooling with its networked neighbors.

independent of the behavior of the ML model itself, and so all communication with the model itself is performed through high-level stateless communication. An architecture diagram of this framework is shown in **Fig. 1**. To illustrate the simple high-level interface that we propose here, we share example code in **Fig. 4**. In this example, a *Scatterbrained* compute node connects to an existing federated cluster by referencing it by its name. This example shows the power of a federated learning: A compute node can join a community of peers and begin learning with a handful of lines of code.

Discussion

Decentralized learning is a desirable property in many domains, including medicine, reconnaissance and exploration, and learning in resource-austere settings. Decentralization enables compute nodes to learn both independently as well as cooperatively, leveraging community knowledge without excessive data sharing. To take advantage of this emerging technology, decentralized learning algorithms will likely vary

in both compute-graph connectivity as well as in the sharing behavior of individual nodes. We propose that a general, high-level, federated learning framework can satisfy both of these needs while improving codebase consistency and readability.

In this work, we shared our reference implementation, *Scatterbrained*, a PyTorch-compatible federated learning framework that is applicable to both centralized and decentralized learning use-cases. We expect that this open-source software package will enable the community to more rapidly explore the rich space of decentralized federated learning algorithms.

Because the field of decentralized learning is nascent, there remain many considerations and challenges to consider before cooperative learning tools are commonplace.

Data Privacy. Despite the inherent privacy associated with sharing model parameters rather than complete datasets, it is an open question as to whether decentralized federated learning satisfies the requirements of complete data security. In other words, it is still unknown whether an adversary in the federated community could successfully reverse-engineer training data from the transmitted weights of peers alone. Though this property is common with traditional machine learning (13), data privacy is of particular interest in the federated learning community.

We propose that selective node connectivity and weight-sharing behavior is sufficient to ameliorate the ability of an adversary to reverse-engineer training data from a federated learning peer (e.g., avoiding sending and receiving weights to/from the same partner in the same epoch), but further work will be required to demonstrate this to a sufficient degree to trust such tools indiscriminately with sensitive data. Similarly, noise might be introduced to the weights prior to transmission: This will reduce the likelihood of data leakage, but at the expense of performance and collective accuracy (14).

Compute-Graph Optimization. We leave the option of compute-graph optimizations available to the users of the *Scatterbrained* library. We believe that there is a rich opportunity to optimize the graph of node connectivity alongside the training process to improve measures such as the rate of information-gain across the community of nodes or the sequestering of adversarial nodes. Though this work is outside of the scope of this manuscript, the *Scatterbrained* networking layer supports this behavior, thanks to the flexible architecture described above.

Future Directions. Decentralized federated learning is a fertile domain for future research, and it is likely that low-power, low-bandwidth edge-computing resources will be key players in the future of distributed computing. As we look to the future of federated learning, we anticipate that distributed, decentralized approaches will enable new and increasingly efficient computing paradigms to address challenges throughout the machine learning world.

```python
import scatterbrained as sb
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset

model = nn.Linear(16, 2)
create_optimizer = lambda model: optim.AdamW(model.parameters())
train = TensorDataset(
    torch.randn(256, 16),
    torch.randint(0, 2, size=(256,))
)
valid = TensorDataset(
    torch.randn(256, 16),
    torch.randint(0, 2, size=(256,))
)
trainloader = DataLoader(train, batch_size=16)
validloader = DataLoader(valid, batch_size=16)

async with sb.Node() as node:
    async with node.namespace(
        "MyNetwork", model, create_optimizer
    ) as ns:
        await ns.train(10, trainloader, validloader)
        await ns.serve()
```

Import the scatterbrained Python module

Create a neural network model using industry standard libraries without modification

Create a compute node with a predefined model and optimizer. Begin training with the local datasets, and join the network of other nodes in parallel

```python
import scatterbrained as sb

async with sb.Node(
    default_operating_mode=sb.OperatingMode.LEECHING
) as node:

    async with node.namespace(name="MyNetwork") as ns:

        await ns.sync_model()

        await node.serve()
```

Import the scatterbrained Python module

Create a compute node, and choose the starting behavior of the node

Attach to an existing community called *MyNetwork*

Download the model from a peer

Begin the compute node server, to receive or transmit model parameters as specified by the node settings

Fig. 4. A simple example using the *Scatterbrained* library. In this example, we demonstrate Python code to start a new *Scatterbrained* compute node. The first example demonstrates the creation of a new node with a simple neural network implemented in Torch. The second example illustrates a new node joining the existing federated community by referencing its namespace, "MyNetwork." It then connects in "leeching" mode and downloads the model (if it's available) from a peer. This means that the newly connected node does not need to have a local copy of the neural network to begin participating in the federated learning community.

Supplemental Materials

All code and scripts referenced in this manuscript are available at github.com/JHUAPL/scatterbrained.

Bibliography

1. 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, 2017.
2. Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, 2017. Accessed: 2020-09-24.
3. Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, et al. Pysyft: A library for easy federated learning. In *Federated Learning Systems*, pages 111–139. Springer, 2021.
4. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
5. Chenghao Hu, Jingyan Jiang, and Zhi Wang. Decentralized federated learning: A segmented gossip approach. *arXiv preprint arXiv:1908.07782*, 2019.
6. Latif U Khan, Walid Saad, Zhu Han, Ekram Hossain, and Choong Seon Hong. Federated learning for internet of things: Recent advances, taxonomy, and open challenges. *IEEE Communications Surveys & Tutorials*, 2021.
7. Anusha Lalitha, Osman Cihan Kilinc, Tara Javidi, and Farinaz Koushanfar. Peer-to-peer federated learning on graphs. *arXiv preprint arXiv:1901.11173*, 2019.
8. Abhijit Guha Roy, Shayan Siddiqui, Sebastian Pölsterl, Nassir Navab, and Christian Wachinger. Braintorrent: A peer-to-peer environment for decentralized federated learning. *arXiv preprint arXiv:1905.06731*, 2019.
9. Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *arXiv preprint arXiv:1803.05880*, 2018.
10. Zhanhong Jiang, Aditya Balu, Chinmay Hegde, and Soumik Sarkar. Collaborative deep learning in fixed topology networks. *arXiv preprint arXiv:1706.07880*, 2017.
11. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
12. Pieter Hintjens. *ZeroMQ: messaging for many applications.* " O'Reilly Media, Inc.", 2013.
13. Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.
14. Mohammad Naseri, Jamie Hayes, and Emiliano De Cristofaro. Local and central differential privacy for robustness and privacy in federated learning, 2021.