Towards Federated Learning using FaaS Fabric

Mohak Chadha
mohak.chadha@tum.de
Technische Universität München
Garching (near Munich), Germany

Anshul Jindal
anshul.jindal@tum.de
Technische Universität München
Garching (near Munich), Germany

Michael Gerndt
gerndt@in.tum.de
Technische Universität München
Garching (near Munich), Germany

Abstract

Federated learning (FL) enables resource-constrained edge devices to learn a shared Machine Learning (ML) or Deep Neural Network (DNN) model, while keeping the training data local and providing privacy, security, and economic benefits. However, building a shared model for heterogeneous devices such as resource-constrained edge and cloud makes the efficient management of FL-clients challenging. Furthermore, with the rapid growth of FL-clients, the scaling of FL training process is also difficult.

In this paper, we propose a possible solution to these challenges: federated learning over a combination of connected Function-as-a-Service platforms, i.e., FaaS fabric offering a seamless way of extending FL to heterogeneous devices. Towards this, we present FedKeeper, a tool for efficiently managing FL over FaaS fabric. We demonstrate the functionality of FedKeeper by using three FaaS platforms through an image classification task with a varying number of devices/clients, different stochastic optimizers, and local computations (local epochs).

CCS Concepts: • **Computer systems organization** → *Cloud computing*.

Keywords: Federated learning, Serverless, Function-as-a-service, FaaS, FaaS platforms, Neural networks

ACM Reference Format:
Mohak Chadha, Anshul Jindal, and Michael Gerndt. 2020. Towards Federated Learning using FaaS Fabric. In *Workshop on Serverless Computing (WoSC '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Cloud computing with on-demand provisioning of resources through virtualization and scaling has provided a method for applications to run at scale at relatively cheaper costs [14]. However, due to the presence of high-speed communication networks, and containerization, computation, and processing of data can occur across a wide variety of devices. These range from resource-constrained edge devices to modestly priced servers with mid-range resources to expensive high-performance computers with extensive compute, storage, and network capabilities. The introduction of serverless computing, particularly function-as-a-service (FaaS) [2] has made the execution of functions on these heterogeneous devices along with the cloud possible [4].

Heterogeneous devices collect manifolds of data [6] each day. Coupled with deep learning, the data generated by these devices can be used to enable intelligent applications. However, with the imposition of various data privacy legislations such as the European Commission's General Data Protection Regulation (GDPR) [7], and increasing concerns over transmission of raw data to a centralized location for training in the traditional cloud-centric approach [5] there has been a growing interest in federated learning (FL) [15]. FL enables the collaborative training of Machine Learning (ML) or Deep Neural Network (DNN) models and addresses the fundamental problems of privacy and ownership of data. It involves local model training on remote clients followed by global aggregation of the updated model parameters. In this context, any end device can act as a "client" or multiple "clients" can be hosted on the same device in isolation [13]. However, training a shared model for end clients makes management tasks such as creation, deletion and invocation challenging. In addition, communication is a critical bottleneck in federated networks. Extending the FL over a massive number of heterogeneous clients can lead up to slower communication in the network than local computation by many orders of magnitude [13].

Connecting multiple FaaS platforms to form a FaaS fabric provides a seamless way for extending FL to end clients where they can act (train or inference) on local data by running FaaS-based functions, while still using capabilities of the FaaS platforms for management, simplicity of fine-grained functions and the capability of the cloud to scale. In addition, each client can store it's data privately in the Cloud storage and FaaS-based client functions running in isolation on the Cloud FaaS platform can utilize the compute capabilities of the cloud for communicating and training the model efficiently.

Our key contributions are:

WoSC '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- We introduce the extension of FaaS to multiple heterogeneous FaaS platforms: Google Cloud Functions (on Google cloud), OpenWhisk (on a high-end server) and OpenFaaS (on three edge devices), called FaaS fabric working together for managing FL at scale.
- We present *FedKeeper*, a python based tool for efficiently managing FL over FaaS fabric.
- We show the functionality of *FedKeeper* through an image classification task with a varying number of devices/clients, different stochastic optimizers, and local computations (local epochs).

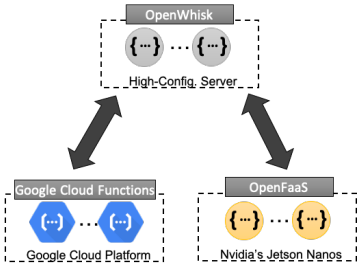


Figure 1. Combination of three FaaS platforms deployed on heterogeneous devices.

2 Background

2.1 Federated Learning

The objective of the standard FL problem is to learn a single ML or DNN model from decentralized data stored on multiple remote clients [15]. A key property of the FL problem is that the training data present on each client does not represent the population distribution, i.e., non-IID. FL system consists of two main components: clients and the FL server. Clients are data owners that participate in a particular round of the FL training process. The FL server is the global model owner.

Training multiple clients through an FL system occurs in synchronous rounds and is a three-step process. In the first step, the server decides the training task, i.e., the global model configuration and the data requirements, and defines the hyperparameters of the training process, e.g., local epochs, learning rate, optimizer. Then, it broadcasts the task and the initial global model weights to the participating clients. The participating clients are decided by the server at the beginning of each training round from a pool of clients and can differ in each round. Following this, in the second step, each participating client uses its local data to update the model weights according to the specified task parameters. Each client trains a local model for the specified number of local epochs using the specified optimizer and learning rate. The updated local model weights are subsequently sent back to the server. Finally, in the third step, after receiving the updated weights from all the participating clients, the FL server aggregates them and sends the updated weights back to the clients participating in the next training round. Steps 2-3 are repeated until the desired accuracy on the test set is achieved or after a specified number of global epochs. In this work, we use the federated averaging (`FedAvg`) algorithm [15] for weight aggregation in the FL server.

2.2 Function-as-a-Service Fabric

FaaS provides an attractive cloud model in which the user is not responsible for server deployment and infrastructure management, but only for writing the code and packaging it. The user implements fine-grained functions connected in an event-driven application and deploys them into a FaaS platform [23]. The FaaS platform isolates the users functions in ephemeral, stateless containers and is responsible for providing resources for function invocations and automatic scaling. Several open-source and commercial FaaS platforms such as OpenWhisk [20], OpenFaaS [19], AWS Lambda [11], and Google Cloud Functions (GCF) [8] are currently available. For commercial cloud providers, the application cost depends on the number of function invocations, memory allocated to the functions, and duration of the function per 100*ms*. Functions can be invoked through a user's HTTP request or custom events created within the FaaS platform. We term a combination of these FaaS platforms deployed on heterogeneous devices and capable of invoking each other's functions as *Funtion-as-a-Service fabric*. In this work, we utilize three FaaS platforms, i.e., OpenWhisk, OpenFaaS, and GCF, shown in Figure 1 as FaaS fabric.

3 Serverless FL Implementation

In this section, we present the overall architecture of our tool *FedKeeper* along with the FL workflow. In addition, we describe the details about the experimental setup for demonstrating FL over Faas fabric.

3.1 FedKeeper

FedKeeper[1] is a client-based python tool for propagating FL-client functions over FaaS fabric. It's main objective is to act as a manager or keeper of various client functions distributed over different FaaS platforms. It has the following responsibilities:

- Facilitating the automatic creation, deletion, and invocation of FL-client functions for each FaaS platform. *FedKeeper* is integrated with the APIs and SDKs of each FaaS platform used in this work.
- Resiliency for FL-client functions. *FedKeeper* keeps track of the functions running on each FaaS platform using activation IDs and automatically creates or invokes the functions which have stopped or failed.

It consists of several sub components, i.e., *Client Register*, *Weights-Updater*, *Client-Invoker* and the *FL-Server*. Each

[1]https://github.com/ansjin/fl_faas_fabric

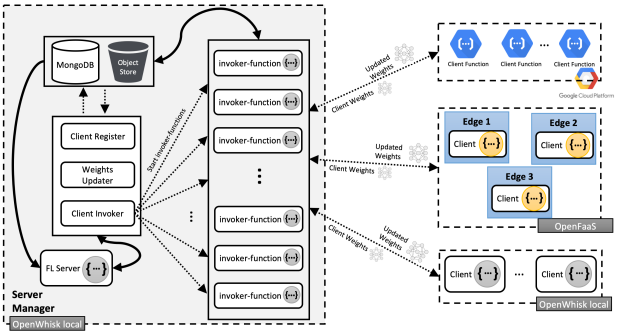


Figure 2. High-level architecture for Federated Learning over FaaS Fabric.

Listing 1. Example parameter configuration file required for invoking a client.

```
{ "client_id": 0,
  "client_type": "edge",
  "url": "client invocation url",
  "model_configuration": {
    "model_type": "nn"
    "input_size": 784,
    ...
  },
  "data_loc":"training data location",
  "training_hyperparameters": {
    "optimizer": "adam",
    "batch_size": 10,
    "local_epochs": "5",
    "learning_rate": "1e-3"
    ...
    }
}
```

component is a function with a memory size of `128MB`, `1GB`, `128MB`, and `128MB` respectively. All the functions have a timeout of `300` seconds and their functioning is described in Section 3.3.

3.2 Experimental Setup

In this work, we utilize a combination of three FaaS platforms, as described in Section 2.2. We deploy OpenWhisk on-premise over a one-node Kubernetes cluster. The node consists of two sockets, comprising of Intel Xeon Gold 6238 processors (Cascade Lake-SP), with 22 cores each and a total of 754GB of main memory. OpenFaaS is also deployed on-premise on an edge cluster comprising of three embedded Nvidia Jetson Nano devices [18]. We utilize k3s [21], a light weight version of Kubernetes to host a cluster on the edge devices. We chose OpenFaaS as the FaaS platform since it provides binaries for ARM processors and because it was not possible to run the heavy footprint OpenWhisk on these boards due to the availability of limited resources. For creating FL-clients on the Google cloud that can be accessed over the internet, we use GCF.

3.3 Serverless FL Worklow

Figure 2 gives an overview of the interaction between different sub-components of *FedKeeper* and the three FaaS platforms in the FL training process. Currently, we assume that the code for the model to be trained is present for each client and is consistent with the model selected by the *FL-Server* for training. We were able to run the framework Tensorflow [1] on all three FaaS platforms. This simplifies the implementation of different models for training. However, due to it's heavy footprint we configured each FL-client with a memory size of `2GB` and timeout of `300` seconds. Initially, FL-client functions are created by *FedKeeper* across FaaS platforms.

On creation of the clients by *FedKeeper*, created client IDs, URLs through which they can be invoked, access authentication required for their invocations, and the FaaS platform on which they were created is stored inside the local Mongo database instance by the *Client Register* sub-component. At the beginning of the training process, the *FL-Server* decides the model to be trained, i.e., `model_configuration` and the specific parameters related to it such as input size, output size, etc. After selection of the model, the *FL-Server* also initializes it and stores initial model weights in the local object store. It also decides the hyperparameters for the training process such as optimizer, learning rate, etc. Following this, it generates a configuration file for each client, an example of which is shown in Listing 1. For the FL-clients present on-premise, we stage the training data locally in the file system on each individual device. On the other hand, for FL-clients present on the Google cloud the training data is stored in each clients corresponding cloud storage bucket. Note that, no other client has access to the data in the bucket apart from the client responsible for it. The generated configuration file is used by the *Client-Invoker* for the invocation of different FL-clients. Depending on the FaaS platform, i.e., the `client_type` and the number of clients in the FL training round, the *Client-Invoker* creates invoker-functions within the OpenWhisk platform. These functions are responsible for the invocation of each FL-client in the communication round through their `url`.

To execute FL-clients in parallel, we do a one-to-one mapping between invoker-functions and the clients. This means that the slowest client will determine the overall time required for the training process. The invoker-functions are also responsible for reading the current model weights from the local object store and forwarding them to the clients. On invocation, the clients compile the model depending on the specified `model_configuration`, set the model weights to the passed values, and update the model weights according to the specified hyperparameters. Following the completion of training for a client, it returns the updated weights back to the invoker-function. The invoker-function updates the weights in the local object store, according to its id and notifies the *Weights-Updater* for its successful completion. *Weights-Updater* keeps track of the clients which have completed training. If any invoker-function returns an error to the *Weights-Updater*, then the corresponding client is invoked again by the *Client-Invoker*. Note that, for momentum based

optimizers such as adam, nadam, the current state of the optimizer is also returned by the client, updated in the local object store, and passed to the client in the next round. On successful completion of all invoked clients, the *Weights-Updater* aggregates the clients weights, stores the aggregated weight in the local object store, and informs the *FL-Server* for the completion of a training round. The *FL-Server* repeats the above process, i.e., invocation of clients and updation of weights until a desired accuracy on the test set of a particular training task is achieved.

4 Experimental Results

In this section, we demonstrate the functioning of FedKeeper for FL with an image classification task. First, we describe the used neural network architectures and the data distribution strategy across clients. Following this, we present convergence results with a varying number of clients, different stochastic optimizers, and local computation. Finally, we present performance results across communication rounds.

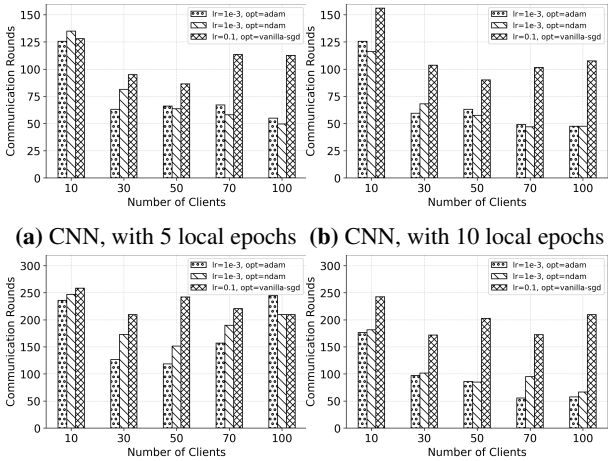
4.1 Neural Network Architecture and Data distribution

We utilize the MNIST digit recognition dataset [12] and use two different network architectures. First, a 2-layer fully connected neural network (NN) with 500 neurons. Second, a convolutional neural network (CNN), with two convolutional layers of kernel size 5x5, followed by a fully connected layer with 512 neurons, and a final output layer with ten neurons. Moreover, each convolutional layer is followed by a max pooling layer of size 2x2. For both network architectures, we utilize categorical cross entropy as the loss function and use Rectified Linear Units (ReLU) [16] as activation functions in the convolutional and hidden layers. Furthermore, we do not use zero padding resulting in 397,510 and 582,026 number of trainable parameters for the two networks respectively.

Similar to [15] and to conform with the non-iid property of data in the FL environment, we first sort the MNIST training dataset with respect to the digit label and then partition it into 200 shards with each shard containing 300 images. Following this, we randomly select and distribute two shards to each client resulting in a maximum of 100 clients for the training process. Out of the 100 clients, we deploy three clients on the edge cluster, i.e., one on each edge device, seven on the OpenWhisk cluster, and 90 on GCF. The data is staged for each individual client as described in Section 3.3.

4.2 Increasing parallelism and local computation

We experiment with the number of clients involved in the FL training process, use of different stochastic optimizers, and the effect of increasing local computation in the participating clients. In all cases, the training task is initially decided by the *FL-server* as described in Section 3.3. We use three stochastic optimizers: SGD, Adamand Nadamin our experiments.



(a) CNN, with 5 local epochs **(b)** CNN, with 10 local epochs

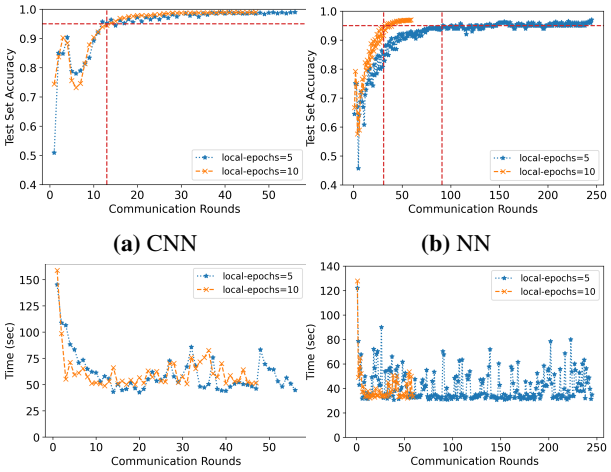
(c) NN, with 5 local epochs **(d)** NN, with 10 local epochs

Figure 3. The number of communication rounds required for reaching 99% and 97% test set accuracy on the MNIST dataset for the two different architectures with a varying number of clients, different local computation, and optimizers.

Through a grid search, we found the best learning rates for the three optimizers to be *0.1*, *1e-3*, and *1e-3* respectively. For the two architectures, i.e, CNN and NN we measure the number of communication rounds required to reach an accuracy of 99% and 97% on the MNIST test dataset. In all our experiments, we fix the batch size to ten and average the results over five runs of the FL training process.

Figure 3 shows the communication rounds required by the two network architectures for different number of clients and with 5 and 10 number of local epochs. For both network architectures, we observe that increasing the number of local epochs leads to faster convergence for a given number of clients and an optimizer. For the CNN network architecture with optimizers adam and ndam, we observe a decrease in the number of communication rounds required to reach the desired test accuracy with an increase in the number of participating clients as shown in Figures 3a and 3b. With SGD as the optimizer, the number of communication rounds required for convergence first decreases as compared to 10 clients but then increases for 70 and 100 clients. A possible explanation for this could be that the initialized learning rate is not optimal for vanilla-SGD. For 5 local epochs with 100 participating clients, we observe a speedup of 2.25x, 2.75x, and 1.14x as compared to 10 clients for the optimizers adam, nadam, and SGD respectively. With 10 local epochs the speedup observed is 2.66x, 2.47x, and 1.46x for the three optimizers respectively. For the CNN network architecture, the optimizers adam and ndam lead to faster convergence than vanilla SGD in all cases.

In contrast to the CNN network architecture, we do not observe a monotonic decrease in the number of communication rounds required for convergence for the NN model with



(a) CNN **(b)** NN

(c) CNN **(d)** NN

Figure 4. The test set accuracy on the MNIST dataset and the average time across each communication round for the two network architectures for 100 clients, with different local computation and *adam* as the optimizer.

increasing number of clients as shown in Figures 3c and 3d. For 5 local epochs, we observe a maximum speedup of 2x (50 clients), 1.64x (50 clients), and 1.23x (100 clients) for the optimizers adam, nadam, and SGD as compared to 10 clients. With 10 local epochs, the maximum speedup observed is 3.2x (70 clients), 2.74x (50 clients) and 1.41x (70 clients) for the three optimizers respectively. In comparison to [15], for 100 clients and 5 local epochs, the usage of adam and ndam as optimizers leads to a speedup of 2.1x and 1.8x as compared to SGD with optimized learning rate. This can primarily be attributed to the faster convergence properties of momentum based optimizers such as adam and ndam.

4.3 FedKeeper Performance

Fig 4 shows the accuracy, average time over communication rounds of the FL training with 100 clients for the two network architectures. The red lines in Figures 4a and 4b represent the number of communication rounds required for reaching 95% accuracy on the test set. For the CNN architecture, 95% accuracy is reached after 13 communication rounds for both 5 and 10 local epochs. On the other hand, for the NN architecture, the desired accuracy is reached after 31, 91 communication rounds for 10 and 5 local epochs respectively. Such an analysis can be useful for trade-off between reaching a desired test accuracy versus overall time required for the training.

For the two network architectures, we observe an average time per communication round to be 60.41s, 41.45s for 5 local epochs and 61.67 and 40.28 for 10 local epochs with *FedKeeper*. In all scenarios, the performance per round is not constant and varies as shown in Figures 4c and 4d. This can be attributed to the performance variation in clients hosted on

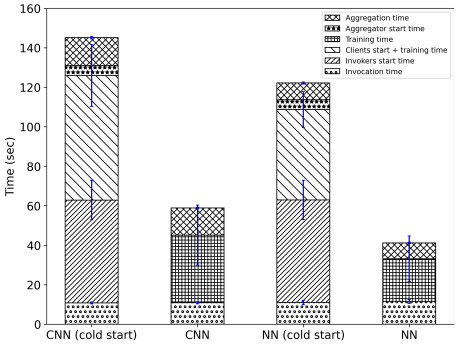


Figure 5. Time distribution for the two network architectures for 100 clients with 5 local epochs and *adam* as the optimizer.

the GCF. The time required for the first communication round is significantly higher as compared to subsequent rounds. This is due to the cold start of functions [9]. Figure 5 shows the distribution of time for the first and subsequent communication rounds for the two network architectures with 5 local epochs. *Invocation* and *Invocation start* time represent the time required for invoking all participating client functions and the time required for all invoker containers in the *FL-server* to start respectively. *Clients start and training time* represents the time required by participating clients to start, train and return the updated weights back to the *FL-server* while *Training time* only represents the time required by participating clients to train and return the updated weights back to the *FL-server*. Finally, *Aggregation start* and *Aggregation* time represents the time required for the *Weights-Updater* container to start and average the updated weights from the participating clients. Note that, *Invoker start*, *Clients start and training time*, and *Aggregation start* times are present only in the first iteration, due to cold start. The average aggregation and training time for the CNN architecture are more than the NN architecture since they depend on the number of trainable parameters in the model.

5 Related Work

Serverless computing is an emerging paradigm which has been shown to support a wide variety of applications such as map/reduce-style jobs [10], linear algebra computation [22], and even applications with performance guarantees [17]. Previous work has also shown how distributed ML training can be supported using serverless functions [3]. Carreira et al. [3], propose Cirrus, a framework to support tasks in ML workflows such as preprocessing of data, model training, and hyperparameter optimization. In contrast to our work, it only supports homogeneous commercial cloud platforms such as AWS Lambda. Morevover, it's worker runtime does not support popular ML frameworks such as Tensorflow [1], which makes the implementation, integeration, and training of different models in Cirrus difficult for the user.

While most previous work has focused on execution of serverless functions on homogeneous systems, Chard et al. [4] propose *func*X, a distributed function execution platform that supports various cloud platforms and modern HPC systems with underlying heterogeneous compute nodes. In contrast to our work, *func*X does not support synchronous training of ML models, and is specifically designed for scientific computing. To the best of our knowledge no previous work in literature has used serverless functions for distributed FL over heterogeneous FaaS platforms.

6 Conclusion and Future Work

We demonstrate that federated learning can be performed on a FaaS-based environment consisting of heterogeneous devices. The computational capabilities of the devices present in the FaaS fabric can be used to optimally schedule FL-based client functions for achieving higher performance. We presents *FedKeeper* for efficiently managing FL over heterogeneous FaaS platforms. Our contributions are:

- **Manageability**: FedKeeper offers easy creation, deletion and invocation of FL-clients.
- **Simplicity**: Model training on individual clients is done using fine-grained FaaS-based functions.
- **Scalability**: FedKeeper offers the capability of running client functions remotely on Cloud FaaS platforms.

In future, we plan to extend the FedKeeper to other FaaS platforms and add security related aspects in it. Furthermore, we will explore techniques to optimize the performance of running client functions in parallel.

ACKNOWLEDGEMENTS

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. Google Cloud credits were provided by the Google Cloud Platform research credits. We thank the anonymous reviewers for their constructive reviews to improve this work and inspire future work.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[3] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[4] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/3369583.3392683

[5] Mingzhe Chen, Ursula Challita, Walid Saad, Changchuan Yin, and Mérouane Debbah. 2019. Artificial neural networks-based machine learning for wireless networks: A tutorial. *IEEE Communications Surveys & Tutorials* 21, 4 (2019), 3039–3071.

[6] Mung Chiang and Tao Zhang. 2016. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal* 3, 6 (2016), 854–864.

[7] Bart Custers, Alan M Sears, Francien Dechesne, Ilina Georgieva, Tommaso Tani, and Simone Van der Hof. [n.d.]. *EU Personal Data Protection in Policy and Practice*. Springer.

[8] Google Cloud Functions. 2020. https://cloud.google.com/functions. Accessed on 09/24/2020.

[9] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[10] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451.

[11] AWS Lambda. 2020. https://aws.amazon.com/lambda/. Accessed on 09/24/2020.

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[13] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine* 37, 3 (2020), 50–60.

[14] X. Li, Y. Li, T. Liu, J. Qiu, and F. Wang. 2009. The Method and Tool of Cost Analysis for Cloud Computing. In *2009 IEEE International Conference on Cloud Computing*. 93–100.

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

[16] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *ICML*.

[17] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-Time Serverless: Enabling Application Performance Guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3366623.3368133

[18] Nvidia. 2020. The performance of modern AI for millions of devices NVIDIA Jetson Nano. https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-nano/. Accessed on 09/24/2020.

[19] OpenFaaS. 2020. https://www.openfaas.com/. Accessed on 09/24/2020.

[20] Apache OpenWhisk. 2020. https://openwhisk.apache.org/. Accessed on 09/24/2020.

[21] rancher/k3s: Lightweight Kubernetes. 2020. https://github.com/rancher/k3s. Accessed on 09/24/2020.

[22] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[23] CNCF Serverless WG. March 2018. Cncf wg-serverless whitepaper v1.0. https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf