Distributed Machine Learning in Future Networks Using Federated Learning

Network Softwarization: Technologies and Enablers

Soroosh Gholamizoj

Abstract

In future networks, the role of Artificial Intelligence, where the intelligence is divided into two levels of global and local, is undeniable. In the local intelligence, it is crucial to have a mechanism in which learning can be done in a distributed and private manner over an immense network of devices. To reach this goal, Google has introduced a new approach in collaborative machine learning called *Federated Learning*. In this work, we train an RNN language model on the works of Shakespeare for a next-word-prediction application using Federated Learning.

Keywords: *Federated Learning; Future Networks; Distributed Machine Learning.*

1 Introduction

In virtue of 5G, the way we interact with objects will be revolutionized. Compared to its predecessors, the fifth generation of wireless technology for digital cellular networks comes with improvements such as higher throughput and lower latency which results in more advanced networks in terms of usage and costs. As a result, the number of devices that can connect to the internet will substantially increase.

At the same time, the Internet of Things (IoT) is going to digitize industries and connect billions of devices around the world to the internet. The growth of these new devices will affect the complexity of managing a network, for the reason that, in such a vast network of connected devices, each node will be a source of data that needs real-time communication and interactions, which means that to utilize these new technologies, working with an extremely high rate of data is expected. Currently, a network is designed and managed by some experts who rely on their knowledge of that specific network and its users. However, with 5G, the network topologies will be more intricate and usage patterns will become less predictable. In consequence, the scale of future networks is on a level of complexity that is beyond the capability of traditional approaches in computer networking for design, deployment, and management.

Meanwhile, Artificial Intelligence and Machine Learning have shown their great power and capacity in recent years. There are various aspects in which artificial intelligence and machine learning can benefit future networks, such as resource management, network monitoring, policy control, service deployment, and operation. Considering this, artificial intelligence and machine learning will undoubtedly play a significant role in future networks.

However, one of the main problems in using machine learning in future networks is the extremely high order of computational resources needed for doing the calculation. In the world of IoT, we will be surrounded by sensors, where each device will be a rich source of data that needs real-time intelligent decisions. With this in mind, managing future networks needs petaflops computing resources, and therefore, having a centralized computing method is indisputably inefficient.

1.1 Distributed Machine Learning

Future networks will have a highly distributed and decentralized architecture where artificial intelligence and machine learning can be added to several layers. Mainly, to bring intelligence into the network, two different levels are being considered: local intelligence, near to endpoints where data is created, and central intelligence, where data is combined to achieve a comprehensive global understanding of networks, services, and functions.

In the local intelligence setting, the majority of end devices are mobile devices (smartphones and tablets). These devices can be considered as the primary sources of data for two reasons. First, because they are built with many powerful sensors such as multiple cameras, GPS, and microphones. Second, is the fact that these devices are being used by users very frequently, anywhere, and anytime. This means that mobile devices have access to excessive data, which in most cases contains private information.

This data can extremely enhance the user experience on the device. However, due to its nature, in most cases, this data is privacy sensitive, very big in quantity, or both. For this reason, it is not possible to use them in the same way as common machine learning

algorithms do, which is to upload and store all the data into a data-center, and then train the model in there.

As these end-devices create enormous datasets, and models get more complex, it becomes more important to have machine learning algorithms in which the optimization of the model parameters is distributed over multiple computational machines. Although some algorithms are proposed for this [1], these algorithms' constraints are only satisfied when the data is IID (independently and identically distributed) over the nodes. In addition, these algorithms have high communication requirements which are not applicable in this setting [2]. Thus, these algorithms are designed for when the training data is centralized, and computational nodes are in a data-center network. However, storing the training data in a data-center has many issues, three of which are: computational complexity, real-time intelligence, and privacy.

2 Federated Learning

In 2014, Google introduced a new approach in collaborative machine learning called *Federated Learning* [3]. The main idea is to exploit the computational power available at the edge of the network to train a shared model, all with leaving the private data at the end-point.

In this setting, there is a central *server*, which coordinates end-point *clients* to collaboratively train a shared model. This process consists of *rounds*, where each round has 4 steps:

- A subset of existing clients is selected. Each client downloads the current model.
- Based on its local data, each client independently computes an update to the current model.
- Each client uploads the model update to the central server.
- The server aggregates these updated models (typically by averaging) to build the next shared model.

This setting is different from other distributed machine learning algorithms that want to scale down common machine learning algorithms to build local models on mobile devices and make predictions on the device by bringing the model training to the device as well. Basically, federated learning only brings the training step of the learning process to the end-devices and keeps the model global and shared. There are numerous reasons for this difference (very large number of clients; highly unbalanced and Non-IID data; unstable and relatively slow network connections) which will be discussed in detail in the following sections.

2.1 Privacy

When talking about a diverse range of devices connected to a global network, one of the most controversial issues that needs consideration is privacy. This problem can be solved with federated learning since no data is uploaded to the cloud, yet the knowledge is being transferred without having any invasion of privacy.

Considering the four *rounds* discussed above, there is no need to upload the local data to the server, and only the update is communicated. Moreover, since these updates are only used to improve the current model, they are ephemeral and can be deleted once they have been used to compute the next improved global model. Consequently, these updates will not be stored in the data center as well.

Despite these improvements, two different cases should be considered. The first case, where an attacker gets the model that is sent from the server to clients. The second case, where an attacker reads the update that a client sends to the server.

In the first case, applying a differential privacy technique can improve the worst-case privacy guarantee [4]. Although, it is worth mentioning that this issue is not specific to this setting, and approaches where private data is stored in the data center and then the model is communicated to end-point for inference also suffer from the same issue.

For the second case, the main solution is to encrypt updates and use standard security protocols for communicating messages. We can also apply the same solution in the first case, where we use differential privacy on each client so that the central server will not be

able to make absolute inferences about a specific client.

In applications where the training is done based on the availability of data on a device, federated learning can considerably improve the privacy, by changing the attack surface from the device and the data-center to only the device.

Considering these into account, federated learning still has significant advantages in privacy over centralized algorithms. Communicating updates, rather than private data themselves, has a higher guarantee on privacy. In addition, the transmitted updates are temporary, and once being used, there is no need to store them in the server.

2.2 Data

The data that is best suited to federated learning have the following characteristics:

- This data should come from a real-world task that is done by users on mobile devices.
- Due to the nature of this algorithm, and its limitations in communication in real scenarios [2], compared to the number of model's parameters, the number of data should be much larger.
- The data should be privacy sensitive, so that uploading the data into a data-center is not desirable.
- If the task is supervised learning, the label should be inferred directly from the user's interaction with the device.

In addition, there is a significant difference between the distribution of the data that we described above and the distribution of the data that is available in most of the datasets stored in a data-center. Here, the data that is used for training and optimization is unevenly distributed over an extremely large number of clients. Accordingly, since different users generate data with different patterns, we can assume that the overall data distribution cannot be represented by the samples of any device. This difference will be discussed more in the next section.

2.3 Optimization

The contrast between data distribution in federated learning applications and other machine learning applications comes from the fact that in many cases, our data comes from a variety of users with different usage patterns. As a result, for the optimization, there are various properties which distinguish this algorithm from other distributed optimization algorithms:

- **Non-IID**: Since for each update the client is only using its own local data, and this data is representing that specific user's usage pattern, the training data used in that update cannot represent the total distribution of the data over the whole population. Thereby, the training data is not independently distributed, which violates the IID properties.
- **Unbalanced**: The clients in our setting are users. Thus, there will be a huge difference in the usage of the service of application between users. This leads us to the fact that some users have more training data, compared to other users.
- **Distributed**: As mentioned in [5], the expectation is that in a realistic scenario, the average number of examples per client is much less than the total number of clients taking part in an optimization.

Some issues may arise once a system is built on federated learning. This algorithm uses real clients for training the model, and these clients are people who are using their mobile devices. Some people may delete a part of their dataset, or they might not be always available which can highly affect the data distribution. Moreover, some clients may never make any collaboration in this process. However, in the `FederatedAveraging`, which is the optimization algorithm based on stochastic gradient descent (SGD) used in this setting [5], it has been assumed that there is a controlled environment where each local dataset is constant and all clients involve in the process of training.

There is a fixed set of clients sized `K`, where at the beginning of each round, a random subset `C` of clients is chosen. As mentioned before, the assumption is that each client has a fixed local dataset. Then, the server sends the latest global model to this subset of clients. Each client uses its local dataset, computes the update, and sends it back to the server.

Finally, the server aggregates these messages and updates the global model for the next round of the process.

$$\min_{w \in \mathbb{R}^d} f(w) \qquad \text{where} \qquad f(w) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(w). \tag{1}$$

Here, $f_i(w)$ is an update that one of the clients sends to the server, and $f(w)$ is the loss of the global model. Thus, we can formulate each update as: $f_i(w) = l(x_i, y_i; w)$ which means the loss of the prediction on example (x_i, y_i) when the model uses parameters w.

Since we have $\mathtt{K}$ clients in each round of training, we define the set of indexes of data points available on each client as P_k, where $n_k = |P_k|$. Considering this, we can re-write $f(w)$ as:

$$f(w) = \sum_{k=1}^{\mathtt{K}} \frac{n_k}{n} F_k(w) \qquad \text{where} \qquad F_k(w) = \frac{1}{n_k} \sum_{i \in P_k} f_i(w). \tag{2}$$

If the dataset was same for all clients, and the training examples were distributed randomly as well as uniformly over clients, then the expectation of a fixed set of examples assigned to a client k would be: $\mathbb{E}_{P_k}[F_k(w)] = f(w)$. However, we have discussed before that federated learning is a Non-IID setting, and therefore, we refer to the case where this expectation does not hold anymore. This means that for a client k, F_k can be a randomly bad approximation to f. In [6], they have discussed federated optimization and optimization for distributed machine learning algorithms in general in more detail.

2.4 Communication Efficiency

In contrast to centralized training algorithms, where the computational cost is the bottleneck, in federated learning, this bottleneck is in communication cost. Since federated learning is a distributed machine learning algorithm, computation is done by clients at the nearly free cost compared to communication costs. However, most clients in this setting have an unreliable and relatively slow network connection which causes the communication cost to dominate.

As reported in [7], one reason for this bottleneck is the asymmetric property of internet connection speeds: "the uplink is typically much slower than downlink. The US aver-

age broadband speed was 55.0Mbps download vs. 18.9Mbps upload, with some internet service providers being significantly more asymmetric, e.g., Xfinity at 125Mbps down vs. 15Mbps up."

The solution that [5] proposes is to decrease the number of rounds of communication before training the model by increasing the amount of computation that is done in each round. They propose two different approaches to this solution. First, to increase `C`, the number of clients working independently in each round. Second, to increase computation that each client performs between each communication round. As they report, the best results are achieved by increasing the computation done by each client, rather than increasing the parallelism.

Thus, since the cost of communication is dominating in this setting, the principal goal is to minimize the number of rounds of communication. To reduce uplink communication cost, [7] proposes two general approaches:

- **Structured updates**: rather than a full model update, learn an update from a restricted space and then parametrize it using a smaller number of variables.
- **Sketched updates**: Compress the update before sending it to the server, where each update is a full model update.

2.5 `FederatedAveraging`

To perform SGD in this setting, one simple approach is one-shot averaging, meaning that each client performs one step of gradient descent for the model based on its local dataset to minimize its loss, and after that, the server produces the next global model by averaging these updated models.

As reported in [5], to solve the federated optimization problem, we can naively apply SGD. To do so, on each round of communication, a single minibatch calculation is done by a randomly selected client. However, the problem with this approach is that to converge to a good model, a very large number of training rounds are required, while as discussed

before, the bottleneck in this setting is the communication cost.

Thus, in the `FederatedAveraging` (or `FedAvg`) algorithm family, the goal is to decrease the communication cost. Three key parameters control the computation of these algorithms:

- `C`: The fraction of clients that perform computation on each round.
- `E`: The number of training passes each client performs over its local dataset on each round.
- `B`: Minibatch size used for the client updates. If $\texttt{B} = \infty$, it means that the full local dataset is used for a single minibatch.

If we set $\texttt{B} = \infty$ and $\texttt{E} = 1$, it constructs an SGD algorithm with a varying minibatch size. Here, on each round, a `C`-fraction of clients is selected. Then each client uses all of its local data to compute the gradient of the loss over them. Therefore, in this algorithm, the *global* batch size is determined by `C`, where if we set $\texttt{C} = 1$, it determines a full-batch gradient descent, which is no longer an SGD.

Algorithm 1 `FederatedAveraging`

Server executes:
  initialize w_0
  for each round $t = 1, 2, ...$ **do**
    $S_t =$ (random set of $\max(C \cdot K, 1)$ clients)
    for each client $k \in S_t$ **in parallel do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $w_{t+1} \leftarrow \sum_{t=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w)**:** *// Executed on client k*
  for each local epoch i from 1 to E **do**
    batches $\leftarrow$ (data P_k split into batches of size B)
    for batch b in batches **do**
      $w \leftarrow w - \eta \nabla l(w; b)$
  return w to server

One simple implementation of distributed gradient descent is that on each client k, we compute the average gradient on the local dataset at the current model w_t, which is

$g_k = \nabla F_k(w_t)$. After that, send this gradient to the server, and the server aggregates them and computes the next update:

$$w_{t+1} \leftarrow w_t - \eta \sum_{k=1}^{\mathrm{K}} \frac{n_k}{n} g_k. \tag{3}$$

What happens here is that each client computes one step of the gradient descent algorithm on the current global model using its local data, then sends it to the server, and the server takes a weighted average of the updates. This leads us to algorithm 1 (`FederatedSGD`), which is first introduced in [5], and is a special case of `FedAvg` algorithms.

3 Implementation

In this work, we want to study how Federated Learning can deploy distributed machine learning in future networks. For the implementation part, we use TensorFlow Federated (TFF) [8] which is an open-source framework for Machine Learning on decentralized data. TFF has interfaces for two different layers: Federated Learning (FL) API, and Federated Core (FC) API. In this study, we work on the higher layer (using FL API) and build an application over the FC layer. The implementation is more focused on an application of FL, rather than the core mechanism of FL. We define a scenario in which we have multiple clients, and we want to train a language model using federated learning.

The implemented model can be used in a next-word-prediction application, for example in a smart keyboard that suggests the next word based on the current sentence and the history of a user's data. This application is suitable for federated learning, because it contains private data, and can have a large number of clients for training the model.

3.1 Data Set

We know that in federated learning, the required dataset is a collection of data from multiple clients, and because of that, in most cases this dataset is non-IID. In theory, as well as in the real-world, for training the model, we are dealing with a very large number of clients, where at a certain time, only a fraction of them are available for training. The state of participating in a training process can be defined as when a client's device is

plugged into a power source, connected to a network, and not being used.

However, in this work, since we are conducting a research scenario, all the data is locally available. Thus, for each round of training, to simulate the same conditions of performing federated learning in the real-world, we can randomly sample a subset of the clients to be involved in. Although for each round the randomly selected subset of clients should be unique, since we know that achieving convergence in this system can take a while [9], for the sake of simplicity, and to avoid running the experiment for hundreds of rounds, we will relax this condition. Instead, what we will do is that we randomly pick a subset of clients once, and then reuse the same subset in each round. By choosing this approach, on the one hand, we will over-fit to a specific subset of clients, but on the other hand, we can speed up the convergence.

TFF has provided a number of datasets in order to smooth the way for experimentation. These datasets are provided in `tff.simulation.datasets` package, and are split into multiple "clients". Here, as mentioned in TFF's documentation, each client corresponds to a dataset on a specific device that might take part in federated training. By using these datasets, we can simulate the challenges of a real scenario, which is to train the model on real decentralized data using non-IID data distribution.

Thus, in our experiment, we use a pre-trained model which is trained on the text from Charles Dickens' "A Tale of Two Cities" and "A Christmas Carol". Then, after importing the pre-trained model, we fine-tune it using federated learning for Shakespeare using a federated version of the data provided by TFF. These datasets are available in `shakespeare.load_data()`, and consist of a sequence of string Tensors, one for each line spoken by a particular character in a Shakespeare play.

3.2 TensorFlow Federated (TFF)

To start the learning process, we use `tff.learning` which is a set of higher-level interfaces that can be used to perform common types of federated learning tasks. For the Federated Averaging algorithm, two different optimizers are implemented: The first one

is `_clientoptimizer`, which is used on each client to compute its local model updates. The second one is `_serveroptimizer`, which is at the server and computes the averaged update, and then applies it to the global model. Since in this setting we are working on a dataset different from standard IID datasets, this two-layer optimizer approach gives this possibility to choose different optimizers, as well as learning rates, for clients and the server.

In order to implement the previously-mentioned processes, two federated-computation properties have been constructed in `tff.templates.IterativeProcess`. This pair of properties are `initialize` and `next`. These computations are programs built based on TFF's internal language that can perform federated algorithms.

First, `initialize` is a function that takes no argument, and returns the representation of the current state of the Federated Averaging process on the server. Second, `next` is a declarative functional representation for the 4 steps that happen in a single round of Federated Averaging: sending the latest global model (which is the current state of the server) to clients, performing a local training (using local data) on each client, collecting and averaging model updates, and producing a new updated model at the server.

TFF provides Transfer Learning, meaning that we can first import a pre-trained centralized model, and then fine-tune it using federated training. This is an important ability, especially in our application, where we can avoid training a language model from scratch, and instead, start from an available pre-trained language model. Thus, in our implementation, first, we load a pre-trained Keras model, and then we fine-tune the model using federated training on a distributed dataset. By doing so, we can still adapt our model to our non-IID dataset.

Considering this, we use a pre-trained RNN model that can generate ASCII characters and then fine-tune it using federated learning. For client optimizers, we use SGD with a 0.7 learning rate, and for server optimizer, we use SGD with a 0.9 learning rate. In the next section, we talk about how we evaluate our model by exporting the final weights to the original Keras model.

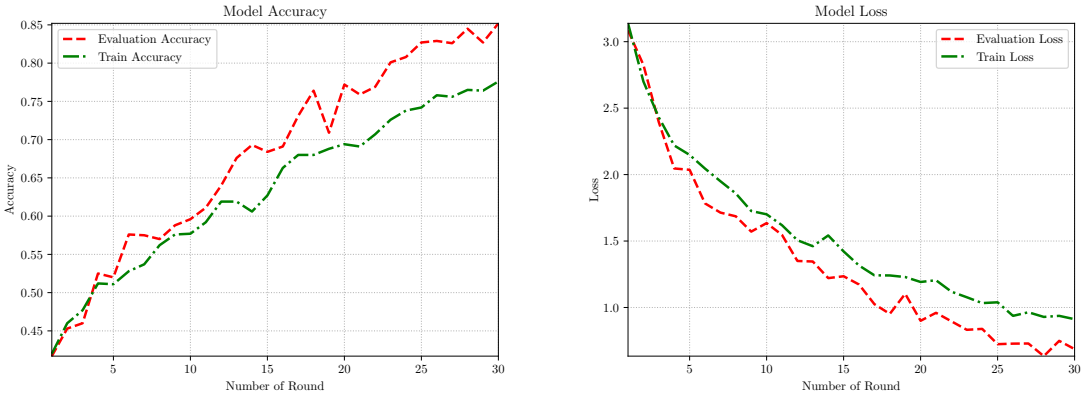

Figure 1: Model accuracy (left) and loss (right) after 30 round of training.

3.3 Evaluation

To evaluate our model, we can use `tff.learning.build_federated_evaluation`, which is another federated computation. To do so, we should pass in the model constructor to this function. This computation is similar to `next`, but with two differences. First, since we are are not going to modify the model in the evaluation step, and therefore the model's state remains constant, this computation does not return the server state. Thus, we can consider it as a stateless computation. Second, because in this step no gradient descent is performed, and thus there is no need to construct optimizers, we can pass in the non-trainable model. That means, evaluation only needs the model, and no parts of the server state related to training, like optimizer variables, are required.

However, for the reason that we are conducting research, and since we have a centralized test dataset in this experiment, we take a different approach to evaluate our model. After each round of federated training, we take the trained weights from the global model, then we feed these weights to the original compiled Keras model, and finally, we use the standard model evaluation in TensorFlow, `tf.keras.models.Model.evaluate()` to evaluate our model on the centralized test dataset. In this way, we can use standard tools to evaluate generated texts. Besides, if there is a centralized benchmark dataset for testing, or for quality assurance purposes, this same technique can be used in a realistic setting.

We run our model for 30 rounds. In figure 1 (left), we can see the model accuracy

increases. For the reason that we start with a pre-trained model, in the first round, the evaluation accuracy and train accuracy are 41.7% and 42.0%, respectively. After 30 rounds of training, the evaluation accuracy is 85.2%, while the training accuracy is 77.6%. The model loss, as it is shown in figure 1 (right), decreases through the training process, which demonstrates the success of learning. From the accuracy and loss curves, we can conclude that more training may result in higher accuracy and lower loss as well.

4 Discussion

In this work, we discussed what federated learning is, and how it can affect future networks, as a distributed machine learning algorithm. We talked about how federated learning is implementing collaborative machine learning without centralized training data, and demonstrated `FederatedSGD`, as a special case of `FedAvg` algorithms, that performs stochastic gradient descent for this setting.

Federated learning is best for applications that are related to private data, and also, where learning can be done over a large network of devices rather than in a data-center. Furthermore, real-time intelligence, which is a key feature for many applications in IoT, can be more feasible in this way. Since there is no need to upload the data, and the learning is done locally, models that are built with federated learning will have lower latency. Although, there are limitations in communication efficiency which can directly affect the total efficiency of federated learning, and need further improvements.

For the optimization algorithm, we assumed that there is a controlled environment where each local dataset is constant, and all clients are involved in the process of training. However, these assumptions may not satisfy in the real world, and thus, it can be a good start point for future works.

References

[1] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng.

Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

[2] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[3] B. McMahan and D. Ramage. Federated learning: Collaborative machine learning without centralized training data. In *Google AI Blog*. 2017. `https://ai.googleblog.com/2017/04/federated-learning-collaborative.html`.

[4] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[5] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *ArXiv*, abs/1602.05629, 2016.

[6] Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, abs/1610.02527, 2016.

[7] Jakub Konecný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *CoRR*, abs/1610.05492, 2016.

[8] 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.

[9] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.