Communication-Efficient Learning of Deep Networks on Decentralized Data

Ashwinee Panda* and Georgy Marrero*

Abstract— **In scenarios where data is generated across many devices, such as data created by mobile applications or on Internet-of-Things devices, centralizing datasets to train or infer machine learning models poses security and privacy risks. Current frameworks for distributed machine learning are not fault-tolerant or especially efficient. We implement an interoperable framework where any model written in Tensorflow or Keras can be trained with edge computing on remote clients. The framework uses federated learning, and is highly fault-tolerant.**

I. PROBLEM DEFINITION AND MOTIVATION

As machine learning models become increasingly ubiquitous, they interact more directly with the consumer. Therefore, they need to be trained more directly on data created by the consumer. Popular real-world examples include filters for inappropriate images that need to be trained on actual pictures (Facebook), text prediction LSTMs that need to learn in online settings to keep up with the evolution of slang (Google), and voice-activated artificially intelligent personal assistants which need to understand how their users react to the day's news to come up with better recommendations the next day (Amazon). However, centralizing these streams of data from millions of people is both an enormous violation of privacy, infeasible from the perspective of storage and bandwidth, and a security liability due to man-in-the-middle attacks. Indeed, many sectors such as healthcare and finance cannot do this for legal reasons, and regulations such as GDPR (General Data Protection Regulation) will make this increasingly difficult. Therefore, we are motivated to create a framework for machine learning models to train on **decentralized data.**

In the motivating setting of machine learning models which make inferences on consumer data, an application with just $1,000$ users each uploading an average of $.65$ photos a day will quickly accrue a dataset of images that motivates the use of deep network approaches such as convolutional neural networks. CNNs and LSTMs achieve state-of-the-art results for image filtering and natural language processing, the latter on datasets as small as the Cornell Movie Dataset of $300,000$ conversations. This is a quantity similar to the data accrued by our toy application in just a year. When this data is multiplied three-hundred-thousandfold to the case of Snapchat, it is clearly attractive to use **deep networks.**

Naive approaches for training neural networks in decentralized settings are training each neuron on a separate machine and that machine's set of data, and training each layer of the network on the data of a separate machine. Neither approach is byzantine-fault-tolerant [1] and also requires each machine to communicate with many other machines when doing each round of backpropagation. Communicating weights repeatedly over these machines is also expensive, as model weight files can reach several hundreds of megabytes in size very easily. This discourages the use of traditional optimization algorithms such as SGD (stochastic gradient descent) wherein each client would need to communicate their update to a central server, wait for a server to perform the computation of aggregating these updates towards the final update, and receive the new model before continuing to train. If this latency is incurred for every update of SGD on a deep model that might need a thousand iterations

*Final paper for CS294-131: Special Topics in Deep Learning of optimization, clearly it would be impossible to train the model in a reasonable amount of time. For this reason, we need an optimization algorithm that is **communication-efficient.**

Distributed machine learning algorithms have existed for some time, in two main flavors: model parallelism and data parallelism. Model parallelism is useful in some niche cases, but not in our setting. Data parallelism naturally fits our scenario, where data is distributed across many devices. The main difference between our motivating setting and the traditional setting for model parallelism, is the differences in dataset quantity/quality and client quantity/quality. 'Super-users' will have many times more data than the average user, and so training a model on just their data would imply overfitting greatly. Most users will not have their data drawn IID, since the local dataset does not represent the larger dataset well. For clients *C* with datasets of average size *d* we will typically expect $C >> d$. Moreover, our approach should be resistant to not only crash-faults, as even distributed Tensorflow implementations such as allreduce are not, but also byzantine faults. In byzantine faults, a user's device can display any behavior, including not sending updates or sending corrupted updates. Finally, the optimization of deep networks in data centers is limited by the computational ability of individual GPUs, as transferring weights over a local network is trivial. By contrast in our motivating setting, we are limited moreso by the upload ability of mobile devices or laptop computers, especially when mobile phones are not often communicating over WiFi connections, than the computing ability of the device, since the devices are computing models over such small datasets.

II. RELATED WORK

A. *Differential Privacy*

Comparatively, the largest efforts in private machine learning are based in differential privacy. Differential privacy aims to make the likelihood of observing a given trend when a specific individual is present in a population equal to the likelihood of observing that same trend when the individual is not present in the population. No aggregate analyses (such as the gradients of a neural net) should convey any information about specific participants in a dataset. Differential privacy can be shown by ensuring that removing a datapoint or data-subset from a dataset does not affect the outcome of an inference. Centralized machine learning on an aggregated dataset, even one with personally identifiable removed (the so-called 'anonymization' of data) still poses significant privacy risks due to the nontrivial likelihood of the data being deanonymized via joins with other datasets.

In the motivating example where the adversary is proposing two datasets S and S with x=1 rows different and Q, a test set of possible outputs of A(), some algorithm, to determine whether A() is evaluating S or S. Q is an interval on T and some constant, where the adversary picks T s.t. When A(?) $\geq$ T, the adversary knows A() operated on S. The learner wants to preserve differential privacy s.t. A(S) and A(S) are epsilon-close, but still have A() be a very good estimate of the . It is commonly achieved by adding noise to datapoints distributed over some prior (commonly Gaussian) to achieve epsilon-differential privacy: $|\frac{\log(Prob[A(S) \in Q]}{Prob[A(S) \in Q])}| \leq \epsilon$.

More formally, Dwork et. al. [4] introduces a variant of epsilon-differential privacy which is epsilon-delta-differential privacy: $P[A(s) \in Q] \leq (e^{\epsilon} P[A(s) \in Q] + \delta)$

Dwork et.al adds noise from a Laplacian distribution that is larger between the gap $A(S) - A(S) = \frac{x}{n}$ where x is most commonly 1. For simple data science questions (averages, conditional inference) the accuracy of A() remains satisfactory even against determined adversaries. However machine learning models that are often given arbitrarily vertically-large (feature-rich) datasets for inferences require that vastly more noise is added to the datasets in order to remain epsilon-differentially private. In our motivating setting, hundreds or thousands of features may be collected on a user -consider a Fitbit. Clearly, adding noise to datasets is insufficient to achieve epsilon-delta differential privacy in our motivating setting, at the scale which we care about.

However, we can apply differential privacy to the setting of Federated Learning to maintain privacy without decreasing the efficacy of our model. Indeed, we will see that the goal of most private machine learning is to maintain accuracy, epsilon-delta differential privacy, and speed. Alternatively, a moments accountant is added to a machine learning algorithm which obfuscates the gradient when it believes that a privacy loss is about to be incurred in the modified setting of Abadi et al.[5] The accountant uses a single-digit privacy budget and is able to achieve high accuracy in the setting of minibatch stochastic gradient descent, which is not dissimilar from the setting of Federated Learning. Some work has been done to explore the possibility of adding noise to each layer of the *model* rather than the data, thus ostensibly preventing the model from just memorizing information about the data, but at the time of writing there is no peer-reviewed material available.

B. Homomorphic Encryption

Homomorphic encryption exists in cryptosystems where computation on ciphertext behaves identically to computation on plaintext. The fastest homomorphic encryption we have observed is in the Gazelle cryptosystem and it still has many unacceptable tradeoffs. Gazelle, while up to 30 times faster than the previous state-of-the-art in homomorphic encryption, is still 20x slower than non-homomorphically encrypted computation.[7]

Gazelle also leaks the model architecture, which is considered crucial information to most machine learning engineers and would be sensitive IP for companies. Although the authors believe that a neural net trained on Gazelle could have enough noise added to it in order to not give away the model architecture (size of filters, order of filters, number of filters) this would of course result in a tradeoff in accuracy, which the authors are already approaching with their existing work.

Homomorphic cryptosystems will typically only support simple operations such as addition/subtraction, multiplication/division. However, one of the main benefits of deep networks is their ability to 'automate' feature engineering and desired complexity through their use of weighted activation functions. These activation functions are not natively supported by homomorphic cryptosystems such as the popularly used Pallier. Different 'homomorphically encrypted machine learning' projects tackle this problem in different ways.

Gazelle attempts to circumvent this by using multi-party computation for activation functions. However, multi-party computation necessitates the use of special protocols for every function whose inputs are shared by its collaborators. Although these special protocols are not difficult to create for the limited set of popular activation functions, they do not bear any resemblance to the popular garbled circuit that is optimized for most modern hardware. As a result, these protocols are many orders of magnitude slower than the simple **max** or **exp** operations we are used to seeing in nonlinearities. Even a garbled circuit generalization approach, which would be hardware-optimized, would have an order of magnitude more operations than a simple max at every step.

PySyft approximates the activation functions of sigmoid and tanh with a Taylor series, which can then be reduced to a set of additions and multiplications that are easily homomorphically encrypted. However, these traditional activation functions have gradients on (0,1) and during backpropagation in a deep network with n layers, n of these numbers are multiplied. The gradient then decreases exponentially in n, making the front layers train very slowly -the famous vanishing gradient problem. This makes PySyft unsuitable for deep learning.

PySyft[11], a homomorphic encryption library intended specifically for machine learning training, is over 100x slower than ordinary training of deep networks, and was designed over months by researchers across the world from companies like DeepMind. The significant slowdown in homomorphic encryption is due to the modular exponentiation. Intuitively, randomness must be added to the problem in order for deterministic attacks to ever have a chance of failing. This randomness must be dealt with, or weeded out later, which costs computational power. After a limited number of multiplications (capped out at 20) it becomes entirely untenable to recover the initial number.

C. Multi-Party Computation

Shokri and Shmatikov[6] presented a similar system to that shown above for multiparty computation of deep learning. This system allows multiple participants to learn neural networks on their own inputs, while benefiting from other participants learning despite not leaking their own inputs. Practical examples exist in the medical sphere, for hospitals who have identically-formatted datasets but are loath to relinquish any control over aforementioned datasets. As in the rest of the related work we surveyed, the purpose of secure multiparty computation (SMPC) is to not leak information about the data.

As we alluded to above, each function that should be encoded in an SMPC setting needs to be written in its own protocol. This makes interoperability extremely difficult, even in the computationally simple settings (taking a max or testing equality between two parties) that SMPC is traditionally used in. In recent years, SMPC's computational efficiency has scaled up due to approaches which try to generalize garbled circuits to much more complex classes for problems. However, matrix-vector multiplications in SMPC are still a far way from being feasible for the swift computing necessary in machine learning.

To create a fully interoperable framework for learning of deep networks on decentralized datasets that maintained the same level of privacy as federated learning, one would need to rewrite the entire of Tensorflow and Cuda to take advantage of thousands of specially engineered SMPC protocols. As Cuda was designed to take advantage of the native computing capabilities of GPUs, there would still be a vast gap in computational efficiency -a true implementation would look like a computer built from the ground-up with logic gates capable of supporting the wide variety of circuits that the library of SMPC protocols would be built on top of. In our motivating setting, this is very impractical, as such a specialized machine could not be expected to achieve the degree of market ubiquity that neural chips, integrated graphics cards, trusted execution environments (Intel SGX) have achieved, which the federated learning approach hopes to draw on for speed and scalability.

D. Inference-Time Privacy

CryptoNets[10] are the most secure and efficient implementation of neural networks which maintain privacy on inference data. However they are not discussed here, as the motivating example -training on extremely secure data from clients phones- does not occur during inference time. Although this is a nontrivial extension of the federated learning approach, we also include a library for inference-time federated learning, specifically for online training on streams of data in a semi-supervised setting, later on in this paper.

III. APPROACH OF ORIGINAL PAPER

The above settings focus on preserving privacy while centralizing data in a datacenter where a network will be trained. This network can still be trained using distributed machine learning for the sake of efficiency, but we will disregard this semantic distinction and refer to these approaches, or the lack of any privacy-preserving techniques, generally as 'centralized machine learning' or 'centralized private machine learning'. In contrast, the main idea behind federated learning is to preserve privacy by distributing a machine learning model over a network to remote clients, bringing the model to the data rather than the data to the model, and then aggregating these trained models, coming to consensus through some consensus mechanism.

Although most prior work dismisses this aggregation (or more aptly, federation) for being vastly inaccurate, federated learning is able to overcome this accuracy by allowing the federation to come to consensus in batches, similar to round-based voting schemes or minibatch stochastic gradient descent. We will take a brief aside to examine the motivation for this approach and why it enjoys a degree of success over centralized machine learning or asynchronous stochastic gradient descent (ASGD).

Training a machine learning model on a user's data specifically is very likely to overfit, as the user's data is non-IID and does not resemble anything close to an unbiased partition of the overarching dataset. In the motivating setting, this overarching dataset would be something like the collective social media interactions of all users in the entire world, and a single dataset would be the Facebook posts of one user of one demographic group in one location. As a result, the network is very likely to learn 'bad traits' by overfitting, such as autocorrecting obscure phrases that only the user says (Google Keyboard), recommending very niche ads (Facebook) or learning to understand a specific accent (Amazon Alexa). However, in learning these 'bad traits' the network will invariably learn underlying 'good traits'. Therefore, each model can be said to have low bias but high variance.

Here we recall two properties of neural networks [2]: for any neural network, the bias can be decreased in exchange for increased variance, and: for any group of networks, the variance can be reduced in exchange for no increased bias. This motivates the approach of combining experts in ensemble averaging, which has been accepted as a simple committee machine for some time. In ensemble averaging, a set of models is generated with different random initializations. Each of them is then trained separately, creating a set of models which each have low bias but high variance. This can be done by overfitting to training data, or just using an unbalanced set of data, or both. The set of models is then combined, and their parameters -typically just the synaptic weights of the network- are averaged. A weighted average can be used, creating a linear combination of networks, but finding a good heuristic for weighting networks can be difficult.

The original paper *Communication-Efficient Learning of Deep Networks from Decentralized Data*[3] which builds on several other papers by McMahant et. al. [8][9] introduces one such algorithm for ensemble averaging that utilizes a federation of clients, called **FederatedAveraging**. Machine learning using this algorithm is therefore **FederatedLearning**.

The Federated Averaging algorithm is a modified ensemble averaging algorithm with a model weighting heuristic defined by the number of datapoints in the training dataset of each model client. It is initially based off stochastic gradient descent. The naive approach we considered above, where a single minibatch gradient calculation is done on a single client every round of communication, requires an incredible number of rounds of communication to reach any reasonable accuracy, but is very computationally efficient as the same amount of computation is done in the decentralized setting as is done in the centralized setting. Instead, in Federated Averaging each client locally does some number of updates of stochastic gradient descent, and communicates these updated gradients back to the server. The server weights each gradient by the number of datapoints that the gradient was trained on, then uses this updated gradient to update a model. The clients pull this updated model and use this updated model to train the next round.

This algorithm can also be considered in the context of a genetic algorithm, where the recombination function is model averaging with synaptic weights, and the fitness function is the heuristic. Either context is just a useful abstraction, and we make use of this abstraction in our extensions to explore different heuristics/fitness functions/consensus mechanisms.

The focus of the experiments detailed in this paper is to illustrate the efficiency and rapid convergence of the Federated Averaging algorithm, in contrast to federated stochastic gradient descent and centralized training. To this end, relatively simple models are trained on well-known datasets, without the use of techniques like dropout, boosting, etc. as these techniques are interoperable and can be applied to all settings. Rather, Federated Averaging is an algorithm for the Federated Optimization setting.

We have previously explored other methods for privacy-preserving machine learning, such as adding noise to anonymized data to maintain differential privacy, homomorphically encrypting data so that it can still be aggregated safely, and using secure multi-party computation so that parties can compute a model over shared inputs without needing to exchange or share their input data. The main difference between the federated learning setting and the aforementioned methods is not transforming the data to maintain privacy. The advantage of this approach is not immediately clear in the setting of two parties, one with a large dataset and the other with a large model, but federated learning quickly shows its superiority when considering the motivating setting.

Transforming the data, whether by adding significant noise in differentially-private training, homomorphically encrypting it, or computing an SMPC protocol over it, and transferring it to a central server, is not practical for edge computing. However, federated learning does not directly compete with differential privacy or homomorphic encryption; when viewing both the model and the data it operates over as 'pieces of sensitive data,' it's easy to see that homomorphically encrypting the model, or adding noise to the weights of the model for the sake of differential privacy, can be combined with federeated learning to product an even more secure scheme. However, we do not explore these approaches in this paper as our goal is to validate the efficiency of the federated learning paradigm and build upon it.

IV. IMPLEMENTATION DETAILS

The original federated learning of Google used TensorflowMobile, optimized to run specifically on mobile phones. However, as we are using federated learning for the purpose of accomplishing decentralized machine learning on the edge of an internet-of-things device database, we will be using standard Tensorflow. We created a model-agnostic application interface (API) where a model written in Tensorflow or Keras can be distributed remotely over many clients with small shards of a collective database. To test this API, we implement a client-server architecture for federated learning in Python, communicating over a local network.

The server begins by randomly initializing a model to the same random initialization M_{init} on all N clients, and distributing the total dataset of size D over these clients. At each communication round/generation/block, the server distributes the current model M to a batch of clients of size $B = C * N$, each client k with a roughly equal partition of the dataset of size d_k. Each client trains the model M on its local dataset to come up with a set of models M_k. These model weights, again represented as M_k are sent back to the server. The server averages these weights to come up with a new model $M*$ according to $M* = \Sigma_{k=1}^{B} \frac{d_k}{D} M_k$ which will be distributed in the next round. We run this process for either a number of maximum rounds, maximum amount of time, or until the process has exceeded a certain maximum time.

Each client is run on the same local machine, and we use Ray for multithreading and concurrency. Ray is a distributed execution engine for machine learning and reinforcement learning.[12] Although the motivating setting would call for massive multithreading with up to 100 threads, we did not have access to this level of computational power without using a GPU (which would have disregarded the point of doing efficient computation over small datasets on remote CPUs in the Internet-of-Things and mobile phone setting) and instead just used the 4 threads available on our laptop computers.

The clients will receive updates from the server, perform E rounds of forward and backward passes on their data in order to progress the model forwards, then send the model away and become dormant. To reduce some computational costs the clients do not become dormant and the same clients are reused in each step, but act on different datapoints in their appropriately-inflated dataset. In this milestone we will not be simulating latency via communication with mobile devices, therefore we will use the same parameters of McMahans paper.

The entire federated learning effort as used by Google makes use of other techniques such as communication-efficient updates via randomized Hadamard rotations and distributed mean estimation, however we will not be implementing these incremental adjustments or using model quantization as the focus of our extension is comparing a different ensemble-learning metaheuristic (validation accuracy) to the approach of weighting by dataset magnitude used in the paper.

We will be evaluating our code reimplementation according to our ability to achieve reasonably similar accuracy bounds, speed of convergence, number of communication rounds, etc. to thresholds of 95% and 99% for 2NN-MNIST and CNN-MNIST, respectively as were achieved in the original paper. Overall, we want to ensure that our federated averaging converges as quickly as centralized stochastic gradient descent.

A. *Hyperparameters*

Learning rate was a very important hyperparameter in the federated learning setting; this was specified in the original paper, and we saw this quickly for ourselves empirically when doing the first few experiments. There are two learning rate hyperparameters in our experiment; the learning rate per client, which we optimized for, and the learning rate decay, which we did not optimize for. The learning rate decay refers to the learning rate dropoff over time, and is enforced after every communication round. The learning rate per client, hereafter just learning rate, was not specified in the paper. We tried briefly to optimize using the Ray-Tune hyperparameter optimization framework, and settled on using a learning rate of $1 * 10^{-1}$.

The fraction of clients selected at each round was also important. This ties into the notion of fault-tolerance, which is especially relevant in the decentralized setting. Essentially, clients cannot expected to always be online, even in the Internet-of-Things setting, and especially not in the mobile phones context. While mobile phones may always be connected to either WiFi or LTE, it is undesirable to communicate large updates over a data network as this will entail passing off significant costs to the consumer creating data. Therefore, our training must be robust even in the face of many clients dropping in and out of the training process. However, we do not allow for clients to drop in and out in our experiments, instead distributing the dataset and then choosing a random fraction of them to train over. We chose two main client-fraction hyperparameters for testing, .1 and .2 as this efficiently displays the tradeoffs in accuracy, time, etc. when doubling the number of clients in the experiment, without using too many clients. Although there are of course increases in these desirable quantities, the high fault-tolerance of the algorithm itself tolerates even just 10% of the clients being online quite well.

Scaling up the amount of computation done per client, or number of epochs of training with the minibatches, is the next important hyperparameter to be tuned after fraction of clients. We tested different numbers of epochs for both the 2NN-MNIST and CNN-MNIST experiments, but settled on 5 epochs of training per communication round per client for the CNN-MNIST and 10 for the 2NN-MNIST. This differs from the optimal hyperparameters of the paper. We believe that the tradeoff may be due to multithreading; we do not enjoy significant performance increases when adding much more computation per client, since we do not have enough threads to fully make use of the increased computation.

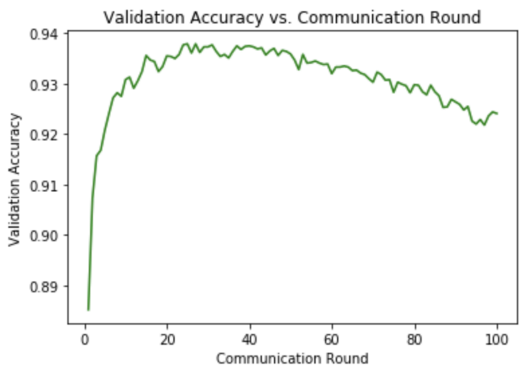
V. RESULTS – REPLICATED

We replicated as many of the results described in the original paper as possible, however some of the experiments were not well-defined or we found it impossible to reproduce the results given computational constraints. We did not replicate the baseline experiments using Federated Stochastic Gradient Descent, as the purpose of those experiments in the original paper was the extremely long amount of time they took, and running an experiment for multiple days was cost-prohibitive. Instead, we only ran the baseline centralized machine learning experiments, and then the experiments using federated learning with the federated averaging algorithm.

We did not replicate the LSTM experiments on the large-scale dataset, because we did not have access to the unique large-scale dataset detailed in the original paper. Some select experiments are produced here, others in the appendix, and the rest in the iPython Notebook in our public code repository. As our **experiment.py** file accepts any number of hyperparameters such as epochs, client fraction, etc. showing a select number of experiments here is fine. Our main goal was to validate the first consensus mechanism/heuristic of weighting by amount of data, and then showcase the increase in fast convergence using the same hyperparameters using our new consensus mechanism/heuristic of weighting by validation accuracy.

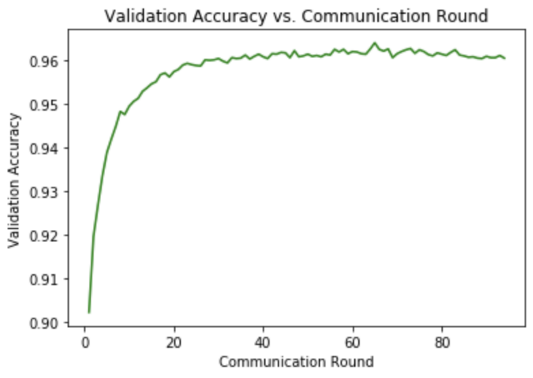
A. *MNIST*

2-layer multilayer perceptron training on MNIST dataset with 100 clients, fraction of clients = 0.1, i.i.d. dataset distribution, minibatch size 50, 10 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



```
Best validation accuracy: 0.9379000067710876 (round: 25)
```

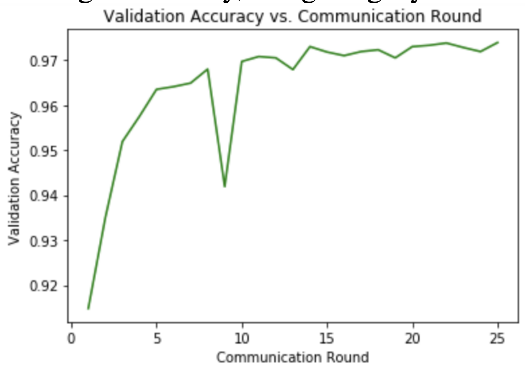
2-layer multilayer perceptron training on MNIST dataset with 100 clients, fraction of clients = 0.2, i.i.d. dataset distribution, minibatch size 50, 10 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



```
Best validation accuracy: 0.9639000296592712 (round: 65)
```

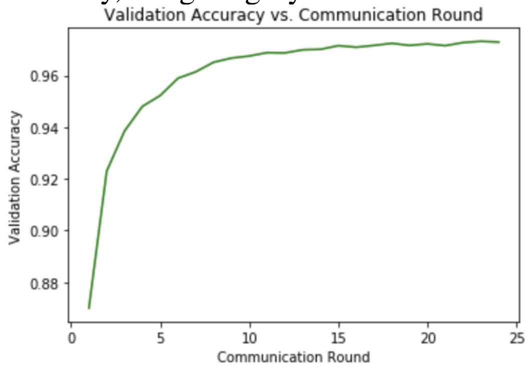
B. *CNN*

CNN training on MNIST dataset with 100 clients, fraction of clients = 0.1, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



```
Best validation accuracy: 0.9739000201225281 (round: 25)
```

CNN training on MNIST dataset with 100 clients, fraction of clients = 0.2, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



```
Best validation accuracy: 0.9732000231742859 (round: 23)
```

C. Analysis

Our results were as expected for both the 2NN-MNIST and CNN-MNIST. We did not attempt to reach the higher accuracy thresholds dictated by the paper, because we were able to see that our approach was correct within 100 rounds for the 2NN and about 20 rounds for the CNN. We were unable to reproduce the results on CIFAR10, as the paper did not specify a learning rate and large learning rates returned NaN errors using Tensorflow but smaller learning rates did not converge in any reasonable timeframe.

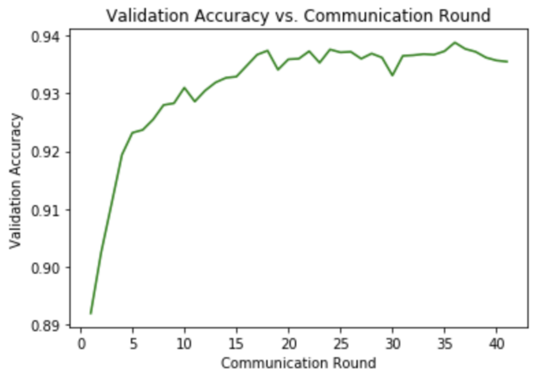
VI. RESULTS – EXTENSION

Our main extension was making the framework for any neural network written in either Tensorflow or Keras to be trained in a federated setting, by creating a wrapper class for both, **TensorflowClient** and **KerasClient**. We implemented the 2NN, CNN and LSTM in the TensorflowClient. We also implemented a Generative Adversarial Network in the KerasClient, but results for its accuracy are not shown here as generally evaluating the performance of GANs is difficult.

One of the arguments in our framework is 'avgtype,' the specific heuristic used to weight each model update when the server is doing ensemble averaging. We generalize that the weighting by dataset approach in the original paper is just one heuristic, and propose two additional heuristics which we empirically validate on the 2NN-MNIST experiments: weighting by validation accuracy and weighting by training accuracy. The former is in line with the ensemble averaging philosophy that we should weight updates to minimize variance, because this comes at no cost in bias. The latter is more experimental, but the underlying intuition is that we can afford to encourage our individual networks to overfit, as the weighted averaging will smooth out our overfitting all the same.

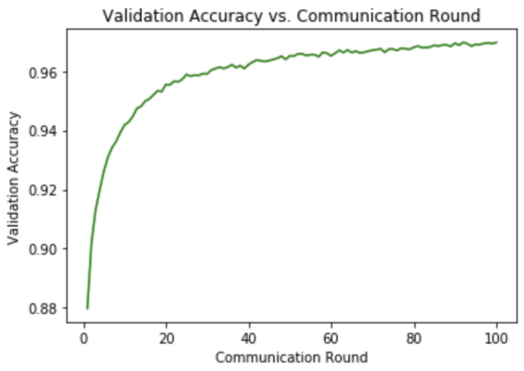
A. MNIST

2-layer multilayer perceptron training on MNIST dataset with 100 clients, fraction of clients = 0.1, i.i.d. dataset distribution, minibatch size 50, 10 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by validation accuracy.



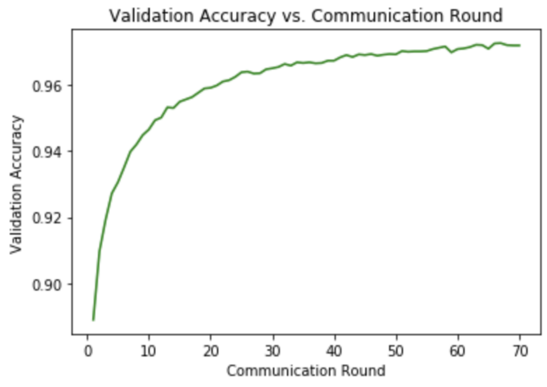
```
Best validation accuracy: 0.9387999773025513 (round: 36)
```

2-layer multilayer perceptron training on MNIST dataset with 100 clients, fraction of clients = 0.2, i.i.d. dataset distribution, minibatch size 50, 10 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by validation accuracy.



```
Best validation accuracy: 0.9697999954223633 (round: 92)
```

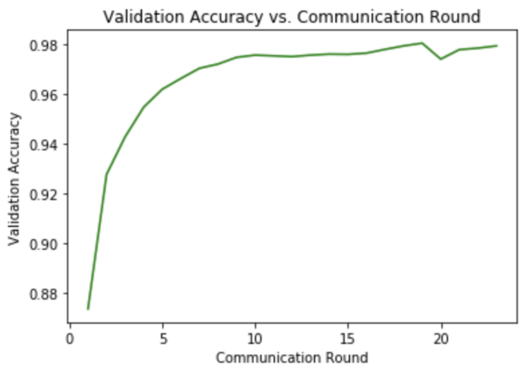
2-layer multilayer perceptron training on MNIST dataset with 100 clients, fraction of clients = 0.2, i.i.d. dataset distribution, minibatch size 50, 10 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by training accuracy.



```
Best validation accuracy: 0.972599983215332 (round: 67)
```

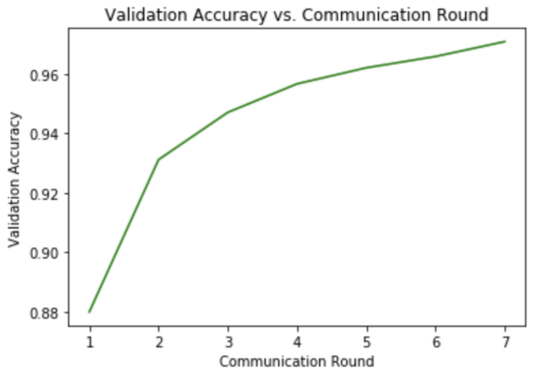
B. CNN

CNN training on MNIST dataset with 100 clients, fraction of clients = 0.1, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by validation accuracy.



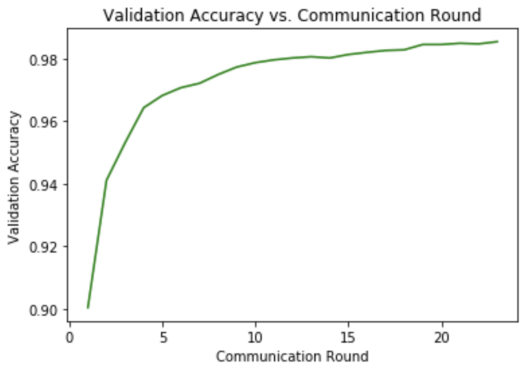
```
Best validation accuracy: 0.9804999828338623 (round: 19)
```

CNN training on MNIST dataset with 100 clients, fraction of clients = 0.2, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by validation accuracy.



```
Best validation accuracy: 0.9708999991416931 (round: 7)
```

CNN training on MNIST dataset with 100 clients, fraction of clients = 0.1, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by training accuracy.



```
Best validation accuracy: 0.9854000210762024 (round: 23)
```

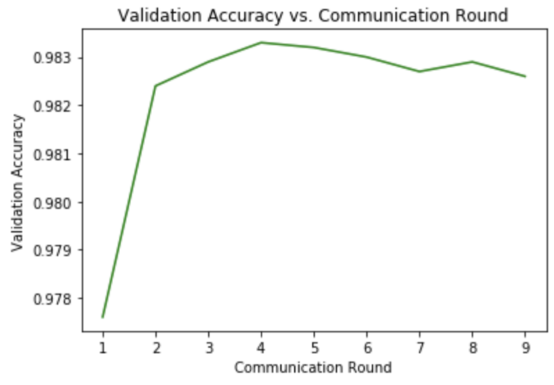
C. Analysis

Our new heuristics performed quite a bit better than weighting by number of datapoints. This is quite intuitive, as in the i.i.d. setting having more datapoints does not significantly impact the likelihood that a model will have achieved the proper accuracy, while the validation and training accuracies are more obvious indicators that the ensemble averaging should prioritize weighting these models.

More interestingly, our models did not continue to improve appreciably (judging by the graphs of validation loss per round) when using the datasize heuristic, but continue to improve significantly even at the maxrounds cutoff of 100 rounds when using both the validation and training accuracy heuristics. This strongly indicates that these heuristics produce overall better optimization algorithms that are more inclined to reach a global max.

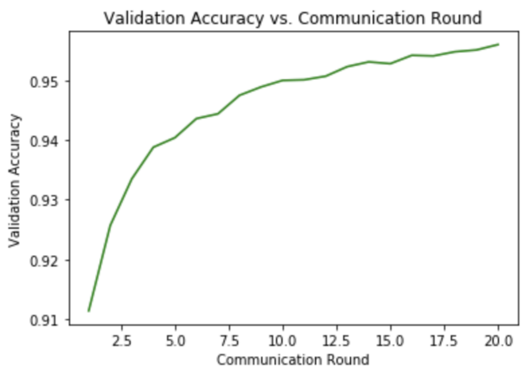
APPENDIX

2NN training on MNIST dataset with 1 clients, fraction of clients = 1.0, i.i.d. dataset distribution, minibatch size 50, 1 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



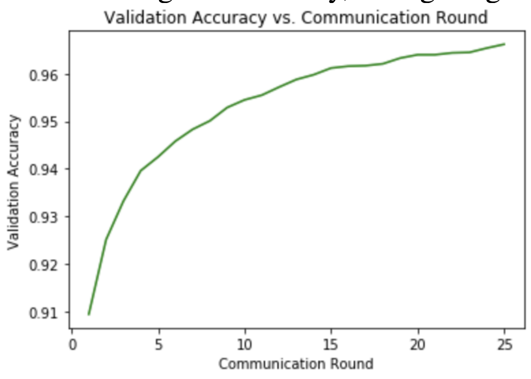
```
Best validation accuracy: 0.983299970626831 (round: 4)
```

2NN training on MNIST dataset with 100 clients, fraction of clients = 0.4, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



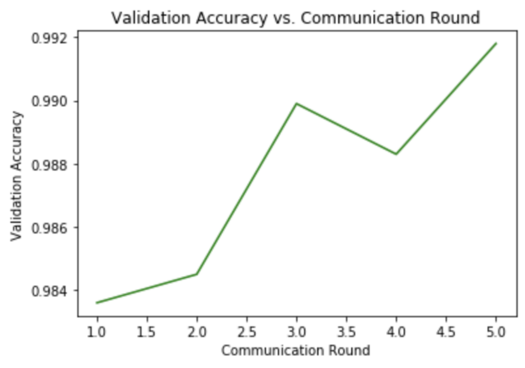
```
Best validation accuracy: 0.9559999704360962 (round: 20)
```

2NN training on MNIST dataset with 100 clients, fraction of clients = 0.7, i.i.d. dataset distribution, minibatch size 50, 5 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



```
Best validation accuracy: 0.9661999940872192 (round: 25)
```

CNN training on MNIST dataset with 1 clients, fraction of clients = 1.0, i.i.d. dataset distribution, minibatch size 50, 1 epochs per client, initial learning rate of 0.1, max 100 rounds, 0.99 learning rate decay, weighting by data.



```
Best validation accuracy: 0.9918000102043152 (round: 5)
```

ACKNOWLEDGMENT

Special thanks to Professor Dawn Song for her continued support of our research efforts into

federated averaging with its various heuristics as a tool for decentralized machine learning.

REFERENCES

[1] El Mahdi El Mhamdi and Rachid Guerraoui. When Neurons Fail, 2017; arXiv:1706.08884. DOI: 10.1109/IPDPS.2017.66.

[2] Naftaly, U., N. Intrator, and D. Horn. ”Optimal ensemble averaging of neural networks.” Network: Computation in Neural Systems 8, no. 3 (1997): 283296.

[3] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson and Blaise Agera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data, 2016, Proceedings of the 20 th International Conference on Artificial Intelligence and Statistics (AISTATS) 2017. JMLR: W&CP volume 54; arXiv:1602.05629.

[4] ”Differential Privacy - Springer Link.” https://link.springer.com/chapter/10.1007/11787006_1. Accessed 19 Mar. 2018.

[5] Martn Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar and Li Zhang. Deep Learning with Differential Privacy, 2016; arXiv:1607.00133. DOI: 10.1145/2976749.2978318.

[6] ”Privacy-Preserving Deep Learning - Cornell Computer Science.” https://www.cs.cornell.edu/~shmat/shmat_ccs15.pdf. Accessed 19 Mar. 2018.

[7] Chiraag Juvekar, Vinod Vaikuntanathan and Anantha Chandrakasan. Gazelle: A Low Latency Framework for Secure Neural Network Inference, 2018; arXiv:1801.05507.

[8] b Konen, H. Brendan McMahan, Felix X. Yu, Peter Richtrik, Ananda Theertha Suresh and Dave Bacon. Federated Learning: Strategies for Improving Communication Efficiency, 2016; arXiv:1610.05492.

[9] ”Distributed Mean Estimation with Limited ... - Research at Google.” https://research.google.com/pubs/pub45672.html. Accessed 19 Mar. 2018.

[10] ”CryptoNets - Proceedings of Machine Learning Research.” http://proceedings.mlr.press/v48/gilad-bachrach16.pdf. Accessed 19 Mar. 2018.

[11] ”PySyft - Library for Secure Machine Learning.” https://github.com/OpenMined/PySyft. Accessed 19 Mar. 2018.

[12] ”Ray - Distributed Execution Environment” https://github.com/ray-project/ray. Accessed 19 Mar. 2018