February 2022 Review

Emmanouil Petrakos

As Machine Learning and Neural Networks become more popular, the demand for additional data is growing. A large amount of data resides in personal devices or servers that are unable to share them due to legal or practical limitations such as privacy laws like GDPR. To mitigate this problem, Federated Learning (FL) algorithms are introduced, in which instead of centralizing data in a server, training gets decentralized by shifting it where the data is stored. The purpose of this work is to identify underlying parallelism in FL training and, if possible, to exploit it. For this to be feasible, the constraints and intricacies of the FL algorithms must first be investigated and defined.

Server - Client System

The FL training is set up to be cross-device compatible. The server holds minimum information about the clients, just enough to maintain connection and communicate. As a result, the server can not address clients directly and no confirmations are conveyed on any direction.

Each global epoch (GE), the server announces the global model to all connected clients. Clients updates the model locally using a part of their data and send back to the server the new trainable variables or their deltas. Finally, the server aggregates the local models and computes the new global model through Federated Averaging. This operation continues until the ending requirements are met, e.g. a desired accuracy was reached or a certain number of epochs passed.

The server, in addition to orchestrating the training and producing the global model, can import a pretrained model and save the final one. Also, in the first GE clients can ignore the global model if it is not pretrained. Instead they can use their own pretrained or initialized model.

Server Overview

Before FL starts, the server has to complete three tasks. Load a pretrained model if it exists, set up a listening socket to receive incoming connections and create a structure to orchestrate the FL training. This structure follows the event driver TCP server paradigm. A single process, handles every event (accept a connection, data available to read or write) on a callback. [1]

An event can be raised either by the listening socket or by a connected one. In the first case, the connection is accepted and a new socket is created and inserted in the event structure. In the second case, the corresponding client either disconnected or data was received. If the former is true, the socket is removed from the event structure and destroyed. If the latter is true, the received data is temporary stored to a buffer designated for that connection. This is necessary because the size of a message may exceed the operating system's socket windows, resulting in multiple read events per message. When a message is fully received, if valid, its model is aggregated to a cumulative model of the local ones.

After any event, the requirements for a new epoch are examined. Their parameters are the number of local models successfully received, the number of connected clients and the number of clients still working. Provided they are satisfied, a new global model is created and broadcasted to randomly selected clients chosen to take part in the next epoch. The new model is calculated by dividing the cumulative model by the number of the local models that were used to create it.[2] Furthermore, the ending requirements are checked and if they are fulfilled, the final model is send to the clients and the server shuts down. The system has been stress tested with random client disconnections and works as expected.

[1]Till now, the server's computations does not seem to add any significant delay. In case that stops being truth, the structure can be modified to assign a thread per client, use a pool of worker threads or just optimize the time consuming sections.

[2]In case deltas are used, the cumulative model is added to the previous global model.

Client Overview

At start up, the client needs to complete two tasks. First, set up the training environment. As training is achieved with Tensorflow in Python and the rest of the codebase is developed in C++[3], the Python interpreter must be embedded before the model is compiled. The second task is to create a socket and establish a connection with the FL server.

After the necessary initializations are completed, the client waits to receive a message by the server. When this happen, the model variables are passed to the Python environment and inserted in the local model. After a predefined number of local training, the new local model variables are exported back and used to create a reply to the server. As the model variables are transmitted in C-like arrays, they can not be directly used in the Python environment. To circumvent that, NumPy array metadata are created around them.

Communication Scheme

The server and each client are separate processes with their own memory regions. Data must be shared efficiently and reliably, hence a fitting communication mechanism is required. This is accomplished by using messages with predetermined size and structure in order to reduce the frequency and complexity of the communication. The messages are C-aligned arrays and their structure is as follows.

Server to client message
flags
global epoch
global model variables ...

Client to server message
global epoch
loss
accuracy
model variables / deltas ...

The flags field is used to give orders or supplementary information to the clients. These are:

- The received model is random and can be ignored.
- The received model is the final one and no further communication will be accepted by the server.
- Evaluate the current global model in addition to training the model

The GE field is required for the server to determine whether a received model is addressed for the current GE and to accept or ignore it. The loss and accuracy fields can be used for more complex algorithms like disregarding local model with inadequate accuracy or greater loss than the previous GE.

At the start of each GE, the server sends a message to each client chosen to take part, which is essentially an order to begin working. When a client finish its task, they answer with a single message, indicating that they have completed their work. There is no need for any additional communication, such as confirmation or synchronization messages. In total 2 * N messages are sent every epoch, where N is the number of the clients. As far as the server is concerned, the clients are stateless and every interaction with them is unique. Each message is independent from the rest and must be self sufficient.

Experiment 1

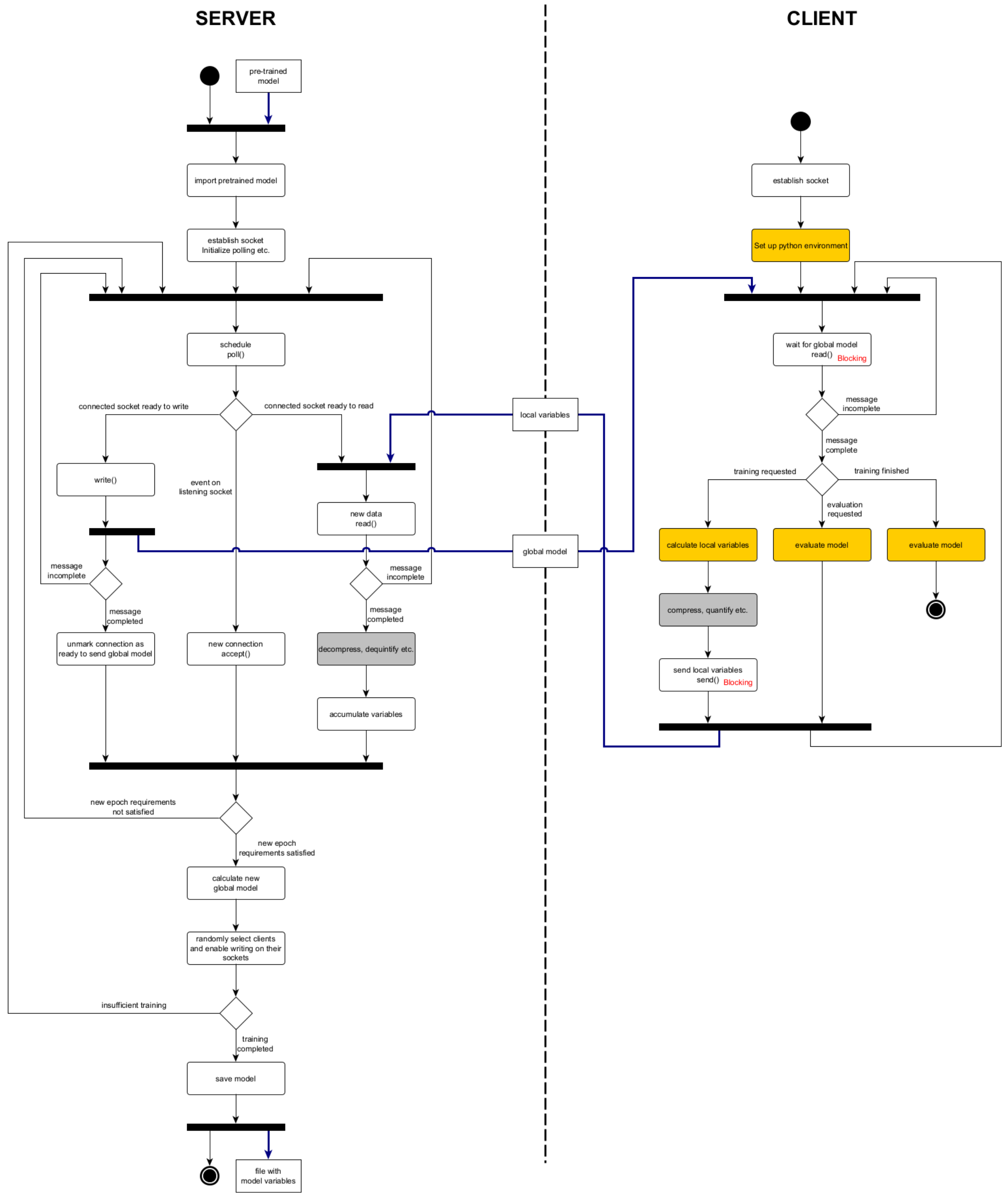
This experiment tests the algorithm with IDD distributed data. A simple CNN that classifies images of the Fashion MNIST dataset and the Adam optimizer are used. The neural network consists of approximately 420,000 Float32 trainable variables and produces messages of 1.7MB. As the server and all the clients are running on the same machine, communication takes place on the operating system's loopback and no substantial timing can be made. The following experiment focus on model accuracy and the convergence rate of the FL algorithm in comparison to locally train a model. The training dataset, which consist of 60,000 images, is split equally between the clients. The following set of parameters and hyperparameters are set.

parameters	
local epochs	1
steps per epoch	3
batch size	10

hyperparameters	
global epochs	500
clients	4

[3]The client's architecture is agnostic of the underlying functions and their implementation, i.e. it can perform any federated operation with any implementation as long as its interface is followed.

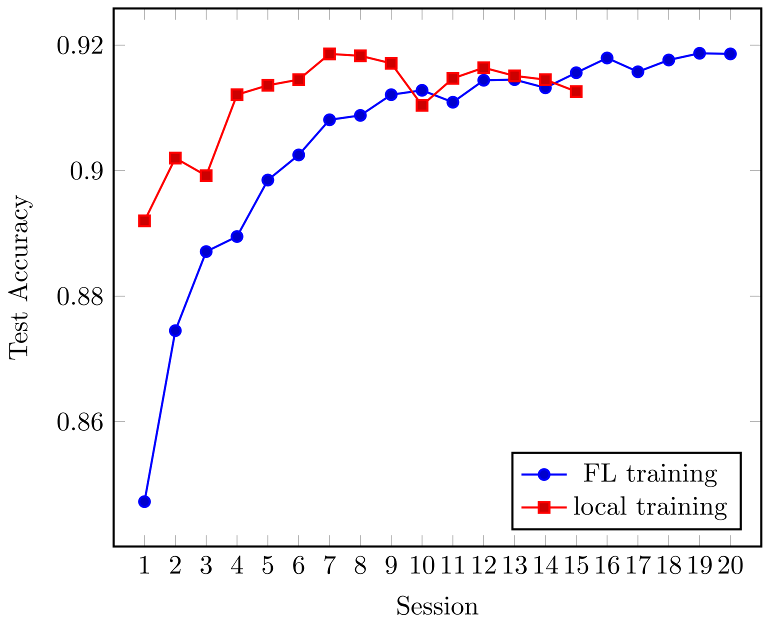
Server - Client Activity Diagram


SERVER
CLIENT
pre-trained model
import pretrained model
establish socket Initialize polling etc.
schedule poll()
connected socket ready to write
connected socket ready to read
write()
event on listening socket
new data read()
message incomplete
message completed
unmark connection as ready to send global model
new connection accept()
decompress, dequintify etc.
accumulate variables
local variables
global model
new epoch requirements not satisfied
new epoch requirements satisfied
calculate new global model
randomly select clients and enable writing on their sockets
insufficient training
training completed
save model
file with model variables
establish socket
Set up python environment
wait for global model read() Blocking
message incomplete
message complete
training requested
training finished
evaluation requested
calculate local variables
evaluate model
evaluate model
compress, quantify etc.
send local variables send() Blocking

Error states are not showed. Yellow states are modelled with TensorFlow and grey are unimplemented.

With these parameters and 4 clients per GE, all data will be passed once every 500 GEs. For the sake of clarity and to not be confused with the global and local epochs, each pass through all data is called a Session.

The accuracy of the FL trained model is compared with the accuracy of a locally trained model. This model has the same batch size, and steps per epoch equal with the number of the data divided by the batch size. Thus, an epoch equals a Session. The following diagram shows the accuracy of the two models on the test data for each Session.



Experiment 1

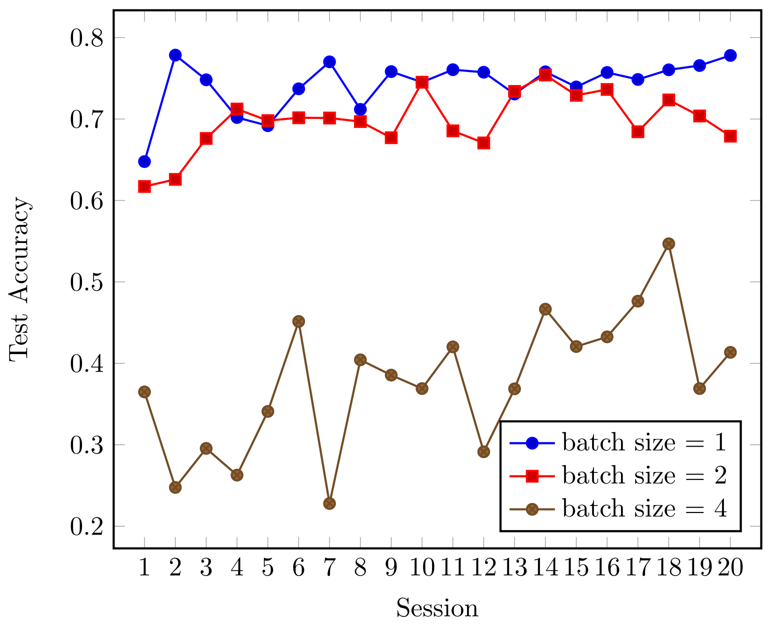
According to the diagram of the experiment, the FL trained model achieves the same accuracy as the locally trained model, albeit at a slower rate. The first model is updated every 150 examples while the second one every 10 examples. According to FL theory, the fewer are the examples that are used every GE, the faster the convergence is. Although, in that case more global epochs are required and the communication overhead increases. In contrast, if more examples are consumed each GE, less communication is required but the convergence rate is smaller and occasionally the model might not even converge. The original FL algorithm without deadline limitations achieved accuracy of 0.92 for Fashion-MNIST with IID data distribution which is comparable with the above results. Experiments with different parameters seems to corroborate this.

Another observation from the above diagram is that FL training combats over-training in some degree. In the locally trained model, the training loss decreased with each step, however this did not translate to higher test accuracy. In contrast, the averaging of the FL algorithm can incur a rebalancing effect and pull out the local models from local extremum. During training the loss of the client models was not always decreasing.

Experiment 2

In this experiment, the algorithm is tested with heavily unbalanced non-IID data and the previous neural network architecture. The fashion MNIST dataset is used and distributed between five clients. The first client holds all the examples with labels 0 or 1, the second client holds all the examples with labels 2 or 3 etc. This is an extreme case of unbalanced data, clients hold no knowledge of the other classes and if they train the network by themselves, a maximum accuracy of 20% is achieved. In such cases, according to FL theory, better accuracy is achieved by minimizing the batch size and examples used per GE.

For this experiment batch sizes of 1,2 and 4 are used. The local epoch and steps per epoch are set to 1. The global epochs are set in such a way that all data are passed once every session.



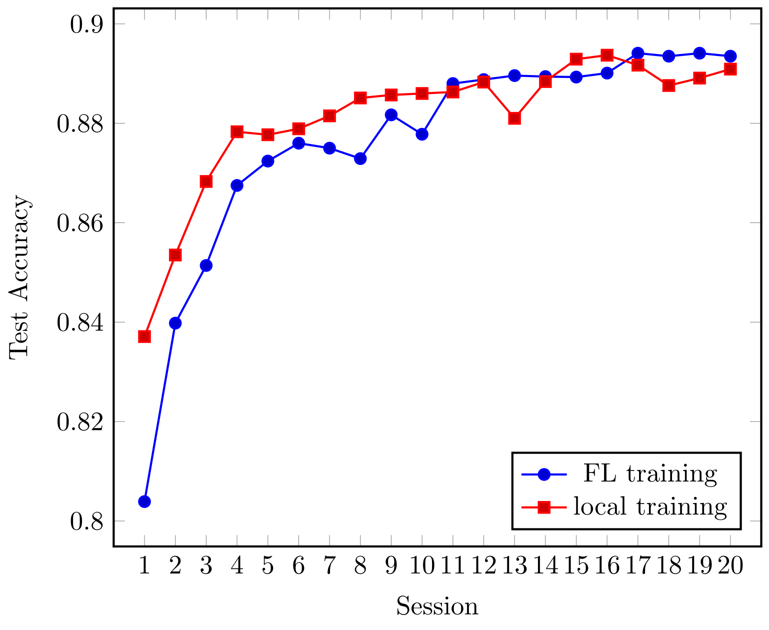
Experiment 2

From the above diagram it appears that the algorithm follows FL theory. Training with batch size 2 or 4 does not converge, while with batch size 1 accuracy seems to follow a stable positive trend after a few sessions. In addition with reducing batch size, better accuracy can be achieved with operations like data rebalancing or using a large pool of clients. The results are comparable with previous works[1].

Experiment 3

In FL training, it is typical to have more clients available than needed during some epochs. Then, the optimal strategy is to use a subset of them to economize in training examples and keep a consistent rate of data consumption. This functionality is implemented and then tested by integrating the LeNet5 model. Eight clients are connected, but only three randomly selected are used each epoch. Each of owns a shard of the fashion-MNIST dataset, which contains 1/8 of the total number of samples.

Because FL training necessitates multiple runs through the data, the risk of overtraining is increased. To combat this, data reshuffling is introduced. When all of the data in a client possession are consumed, they get reshuffled and new batches are created. This is also used during local training in this experiment. The batch size is set to 20 and the steps per epoch to 2. Additionally, 10000 GE are required to consume the data 20 times during training.



Experiment 3

According to the diagram, The algorithm operates normally and reaches similar accuracy with training locally. Also, no signs of overtraining are present.

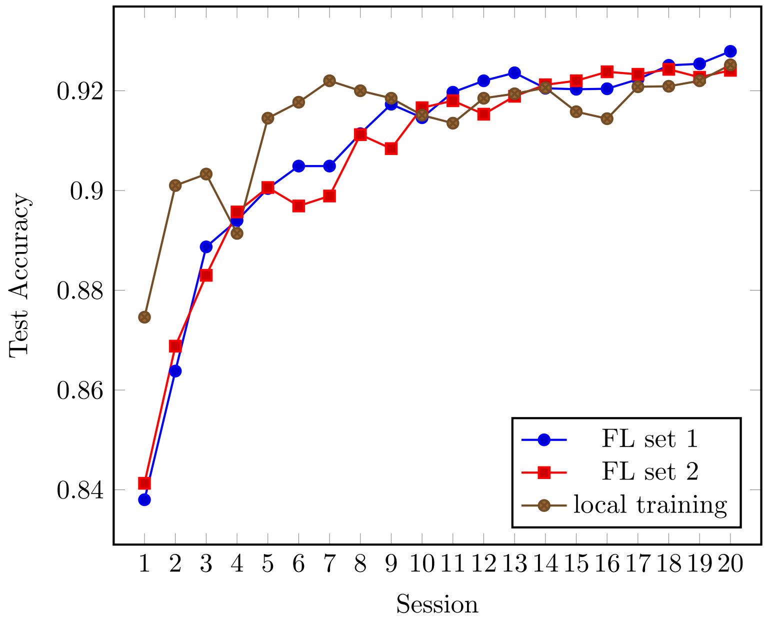
Experiment 4

In this experiment, a CNN model architecture is implemented, which utilizes the inception module from GoogLeNet. It consists of 277082 trainable variables and uses the Adam optimizer. Furthermore, the data is IID distributed and data reshuffling is included. The first goal is to assess the response of the FL algorithm with more complex structures.

Two sets of parameters and hyperparameters are used and are detailed in the following array. In both cases, the data is equally divided between 5 clients, but only 3 are used each epoch. The difference between the sets is how many examples are consumed each GE. The second goal of the experiment is to assess the impact of the data utilization per epoch.

parameters	FL set 1	FL set 2	local training
clients per GE	3	3	1
steps per GE	1	2	examples/batch size
batch size	20	20	20
examples per GE	60	120	all
GEs to use all examples (Session)	1000	500	-

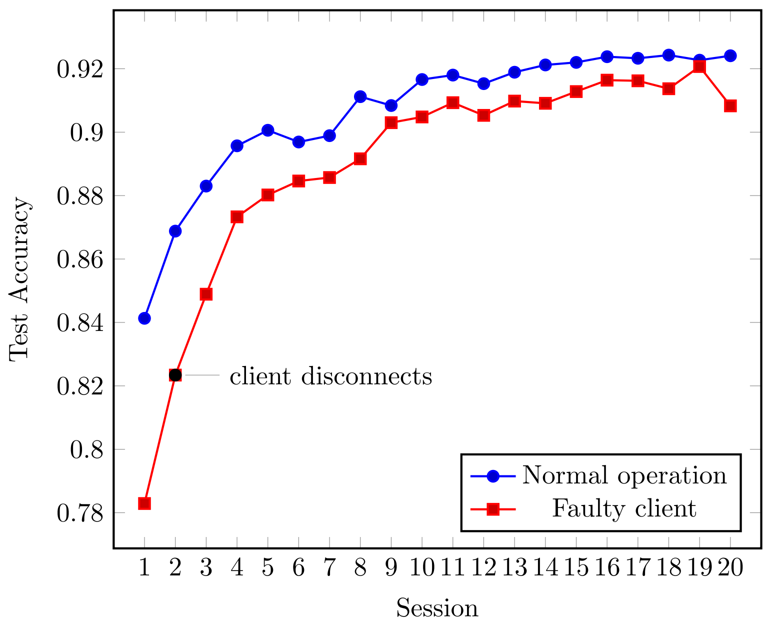
The results show that both FL sets reach satisfactory accuracy, albeit at a slower rate than local training. While the second produce slightly less accuracy, it is worth noting that it requires half of the communication of first set. This has the effect of doubling the computation to communication ratio and being a more lucrative target for parallelization. Experiments with larger batch size and steps per epoch should be conducted.



Experiment 4

Experiment 5

In an edge environment, the clients tend be unreliable and any algorithm must be resilient to random faults. To simulate such a case, the second setting from the previous experiment is used with one extra client, which gets disconnected 1/10 into training. That means for 90% of the training, there is not access to 1/6 of the data. All other parameters remain identical.



Experiment 5

While the accuracy of the model suffers, the effect is not disastrous as training continues and accuracy is still in acceptable range. In a real-world scenario, such a problem can be solved by deferring a portion of the training in a time when new data is available from clients.

Future Work

- Experiments with greater batch size and more steps per epoch should be carried out.
- Larger models (ex. AlexNet) with up to 50 million variables have been successfully integrated with the FL training algorithm, however the resource requirements are prohibitive for training in a reasonable amount of time. Find a medium sized model that stresses the algorithm but its training duration is not excessive.
- Experiment with a larger pool of clients. Test with up to 10 clients have been carried out so far, however the memory requirements of instantiating the Python interpreter and Tensorflow runtime numerous times are prohibitive.
- Experiment with different sets of parameters, hyperparameters and optimizers. Till now all experiments have been conducted with the Adam optimizer.
- Extend the new epoch requirements to consider time passed in the current GE, in order to discard stragglers. The timeout of the event structure can be used. In addition, connections that have not send any data for a few consecutive GE could be dropped.
- In addition of importing a pretrained model, the server could be able to partially train a model locally, before the FL commences.

Notes

1. https://arxiv.org/pdf/1804.08333.pdf