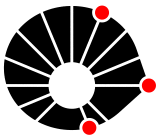

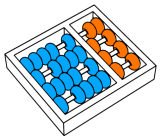

Vertical Federated Learning for Emulation of Business-to-Business Applications at the Edge

C. Avelar *J. V. Gonçalves* *S. Trindade*
N. L. S. da Fonseca

Technical Report - IC-21-04 - Relatório Técnico
February - 2021 - Fevereiro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Vertical Federated Learning for Emulation of Business-to-Business Applications at the Edge

Carlos Avelar* João Vitor Gonçalves† Silvana Trindade
Nelson L. S. da Fonseca

Abstract

Given the ever-increasing constraints and concerns regarding data privacy and sharing, a method to train collaborative machine learning models without exposing training data can become a major part of the way that data science is done. In this work, we illustrate the concepts of Vertical Federated Learning, along with a practical implementation emulating a real scenario of collaborative training of a model. We evaluate the cost associated with homomorphic encryption that enables Federated Learning approaches and show results of an MNIST solving model using Vertical Federated Learning.

1 Introduction

Recently, there have been numerous Machine Learning (ML) techniques that have seen a growth in interest and have been applied to problems. The query "machine learning" on Google Scholar returns more than 350,000 hits when filtering for only 2019 articles.

Although these algorithms have been greatly improved, the biggest leap happened in supervised methods, making the limiting factor of applications, the lack of labeled data for training.

This is not only true in areas in which labels are scarce or that there are little data to work with, but also in areas where privacy is fundamental, like healthcare, industry 4.0, and finance applications. One of the problems is the exponential growth of model size, which increases the time and resources necessary to train and evaluate these models[10].

The usage of distributed training techniques is not only desirable for faster training but necessary because the limitations of hardware and software at the edge, since edge nodes (edge server and end devices) cannot individually handle the training of larger models. And as consequence, distributed training has been very optimized already.

Traditional ML algorithms are executed without restrictions to data access, however when the focus is data privacy and protection a different approach is needed. With this in mind, Federated Learning (FL) paradigm was introduced by Google [3], combining distributed learning with privacy-preserving techniques, allowing the same model to be trained on private distributed data.

One of the first examples of Federated Learning (FL) [3] in production was released by Google, with the Android Keyboard autocomplete [4]. It is more specifically an example of Horizontal Federated Learning (HFL), so it distributes the training across millions of devices and aggregates the model in a central server, proving that the technique could be used for large scale B2C training. However, the HFL assumes that full samples of data and labels are distributed across the clients, which may not be the case in Bussiness-to-Bussiness (B2B) scenarios given the fact that usually,

*Inst. de Computação, UNICAMP, 13083-852 Campinas, SP. c168605@dac.unicamp.br
†Inst. de Computação, UNICAMP, 13083-852 Campinas, SP. j176353@dac.unicamp.br

companies have overlapping datasets with different features, and that is the motivation behind Vertical Federated Learning (VFL) [5]. Here the parties could have only part of the total features that were desired for the model, allowing for a different set of problems to be solved. For instance the datasets of a hospital and a pharmaceutical company would have many different features.

In FML [5] the authors describe algorithms and discuss the differences between the Federated Learning types. In [6], the authors introduce a solution to the asymmetrical datasets problem, when one party has an overwhelmingly big dataset, that might contain the whole dataset of the other party, or when the absence of an entry in the dataset might be a privacy concern. However, these examples use only regressions to demonstrate the method, making it hard to extrapolate some ideas for more complex models like Multi Layer Perceptrons (MLPs) using arbitrary architectures.

In this work, the aim is to emulate Vertical Federated Learning at the edge using ContainerNet network emulator to demonstrate its results when compared to a traditional approach to machine learning with the data and training being centralized; then, measure the associated overhead of encryption on the message exchanges.

In the remainder of this work, we briefly explain some of the important concepts needed to understand VFL in Section 2; in Section 3 the application and the experiments are described; in Section 4, we present the results from experiments, and conclude on Section 5.

2 Federated Learning

Federated Learning (FL) is a method of decentralized machine learning, that allows for training models with multiple edge nodes each containing their local data and without sharing the data among them.

It is mostly useful for applications where we have multiple entities holding private information that can not be shared but is relevant to training a machine learning model. Examples of private data would be sensitive information like medical exams, received email contents, and geographical location.

FL works with network message exchanges between the multiple data holders and a centralized server, which brings a lot of overhead to the training process, where a traditional centralized approach would not have these inefficiencies and therefore would be considerably faster. So FL is only a desirable approach in cases where there is a privacy concern involved and centralized training would not be possible.

The ability to be able to train models without direct access to data can be very powerful, with applications in medical imaging, allowing for multiple hospitals to train models together using patient records, without directly exposing that information.

As mentioned previously, FL was used by Google [4] with the Android Keyboard auto-complete, allowing for better keyboard suggestions without having direct access to the data used in the training process.

2.1 Data Distribution

There are two different approaches to Federated Learning: Horizontal Federated Learning (HFL) and Vertical Federated Learning (VFL); The difference between the two lies in the way the data is distributed (Figure 1).

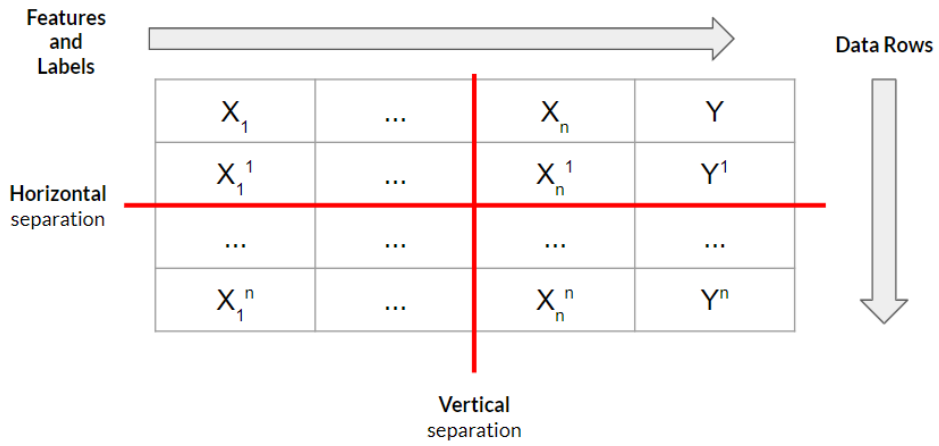

Figure 1: Comparison of horizontal and vertical cuts in a dataset

In HFL the data is horizontally distributed between the parties (Figure 2), meaning that each party has some rows of the data, which include all the features and the label. Here, there is generally no row intersection between parties, only a feature intersection.

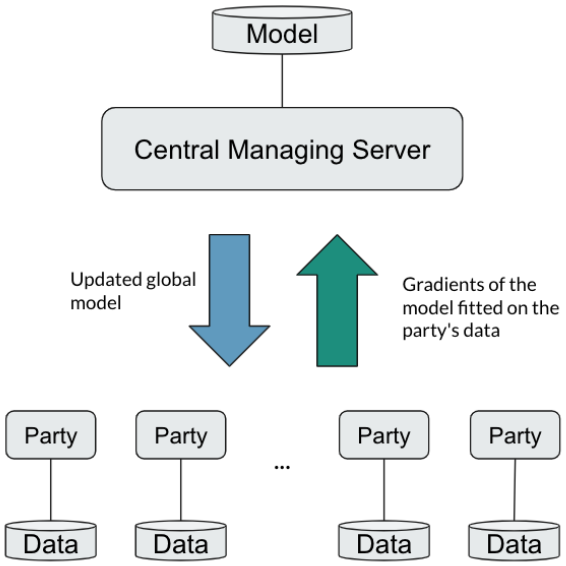

Figure 2: Diagram of Horizontal Federated Learning data distribution

In VFL each entity has a vertical cut of the data (Figure 3), in other words, that is, each one has part of the features and labels. A requirement for the algorithm to work is row intersection between the entities and a set of shared features that can be used to align both datasets called the ID. So there must be a data overlap and we must be able to identify where the overlap occurs.

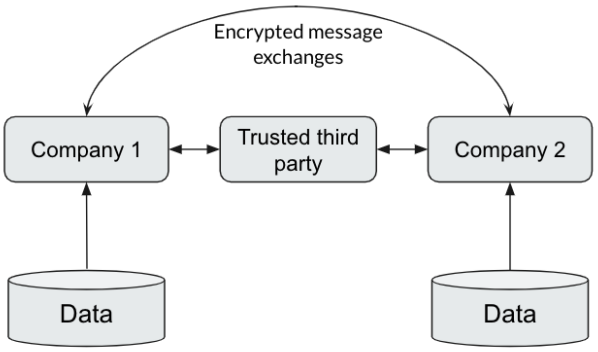

Figure 3: Diagram of Vertical Federated Learning data distribution

2.2 Private Entity Alignment

ID alignment algorithms find intersections for the features (Fig. 4) on the data to create a single dataset out of two or more. Private ID alignment does the same as just described, however in a way that doesn't expose the data being used to the party running the algorithm.

For this work, we have assumed that the ID alignment step had already been done, so further details on ID alignment algorithms will be omitted.

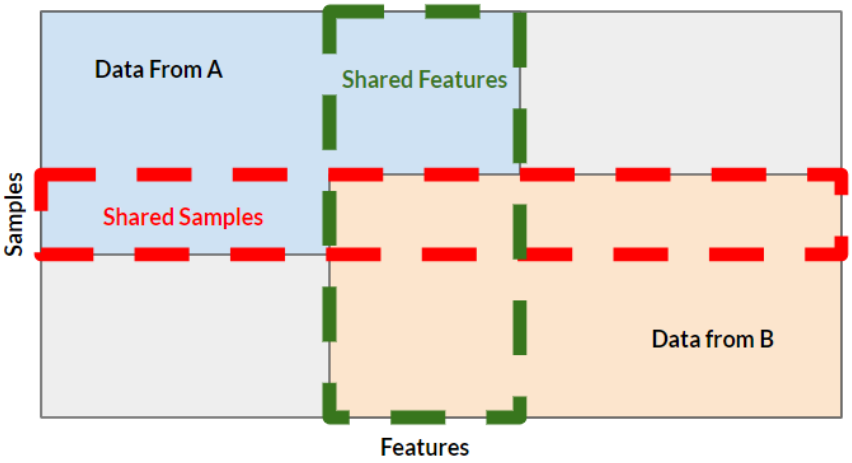

Figure 4: Diagram of Vertical Federated Learning sample distribution

2.3 Homomorphic Encryption

The idea behind Homomorphic Encryption is simple, it allows sum and multiplication operations to be made on cyphertext. So it is possible to sum two cyphertexts A and B and obtain a cyphertext C with the result. Moreover, it is possible to multiply a cyphertext A by a cleartext B and obtain a cyphertext C with the result.

Using {{x}} to denote the homomorphic encryption of x, being A and B numbers, the operations are:

Add operation: we can add two encrypted values and get an encrypted result

$$\{\{A\}\} + \{\{B\}\} = \{\{A + B\}\} \tag{1}$$

Multiply operation: we can multiply two encrypted values and get an encrypted result

$$\{\{A\}\} * \{\{B\}\} = \{\{A * B\}\} \tag{2}$$

Further details of the encryption algorithm can be found in the article [9].

2.4 Vertical Federated Learning

Vertical Federated Learning (VFL) needs three parties to execute, two clients A and B that hold the data and a coordinator C, although the coordinator role can be made by A and B, effectively eliminating the need for the extra party. The initialization follows these steps: first is necessary to align the datasets of the multiple parties using a Private Entity Alignment algorithm; then C creates a pair of encryption pairs and shares the public keys with both clients. With these steps done, the main loop can start. The main loop consists of both clients doing a forward pass on their data; clients exchanging homomorphically encrypted information for calculating the encrypted loss; sending the encrypted loss for the coordinator for decryption; and finally a backward pass.

Now, we illustrate the process on an example (Algorithm 1). The example assumes an arbitrary model, mean quadratic error as loss and that client A has $\{x_A\}$ and client B has $\{x_B,y\}$, x being a set of features and y the labels.

We use {{x}} to denote the homomorphic encryption of x.

Algorithm 1: Vertical Federated Learning Main Loop

initialization
while *not done* **do**
  A: $\mathrm{u}_A = model_A.forward(x_A)$
  A: $\mathrm{L}_A = u_A^2$
  A: send $\{\{\mathrm{u}_A\}\}$ *and* $\{\{L_A\}\}$ *to* B
  B: $\mathrm{u}_B = model_B.forward(x_B)$
  B: $\mathrm{L}_B = u_B^2$
  B: $\mathrm{d} = \{\{\mathrm{u}_A\}\} + \{\{u_B - y\}\}$
  B: $\{\{\mathrm{L}\}\} = \{\{\mathrm{L}_A\}\} + \{\{L_B\}\} + 2\sum(\{\{u_A\}\}(u_B - y))$
  B: send {{L}} and {{d}} to C
  C: decrypt {{L}} and {{d}}
  C: send L and d to A and B
  A: $\mathrm{model}_A.backward(d)$
  B: $\mathrm{model}_B.backward(d)$
end

3 Methodology

We implemented the VFL algorithm using Python 3.9.1, PyTorch[1] for model and dataset handling, and Pyfhel[2] and Numpy[3] for homomorphic encryption.

[1] `https://pytorch.org/`
[2] `https://pyfhel.readthedocs.io/en/latest/`
[3] `https://numpy.org/`

Containernet[4] was used for emulating a virtual network where we can create scenarios of multiple hosts exchanging messages. It is based on Mininet network emulator which is a tool that allows for emulating multiple hosts running application code connected by switches creating an emulated virtual network of hosts. Containernet allows for docker containers to be executed as hosts, simplifying development and allowing for separate file systems between the different hosts in the application.

3.1 Communication

The process communication was achieved using Web Sockets, which allows for a persistent HTTP exchange of messages between hosts, it has the advantage of reducing the overhead associated with HTTP nonpersistent HTTP requests while having a well-defined protocol that helps with compatibility and ease of development.

3.2 Data

The MNIST dataset was used.[2]. MNIST is composed of 60,000 images of handwritten numbers, of which 10,000 are for testing. The images are composed of 28x28 arrays of binary intensity and the images are evenly distributed among the 10 numbers (0 to 9).

In our experiments, we have cut the images vertically in two. That way each half stays with one client and one client will hold the labels. The process is shown in Figure 5.

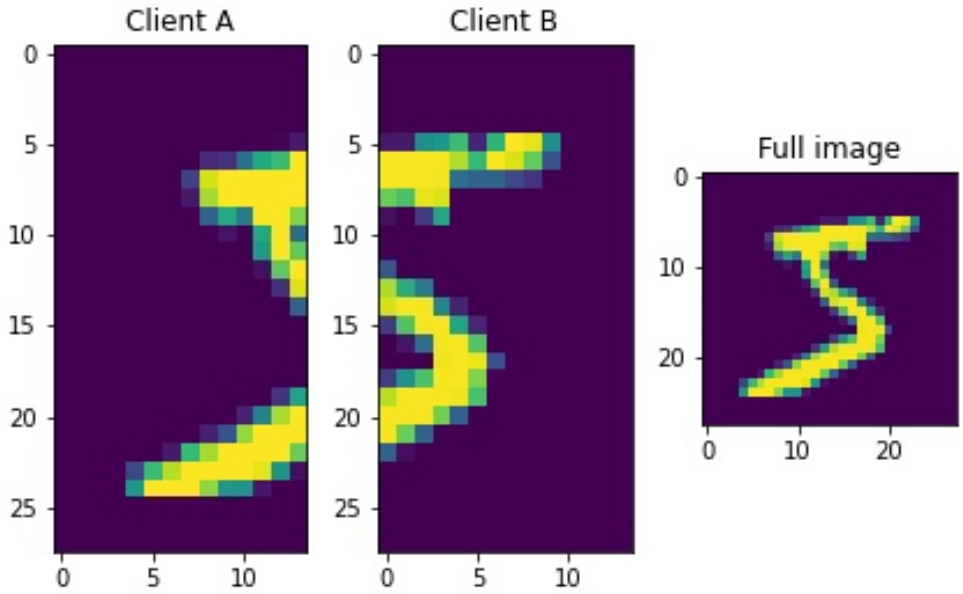

Figure 5: Vertical dataset division of a MNIST dataset sample row.

3.3 Model

The model used for the experiments was a Multi Layer Perceptron (MLP) with the architecture shown in Figure 6.

[4] `https://github.com/containernet/containernet`

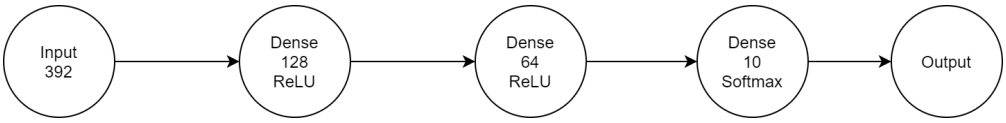

Figure 6: Model architecture

The two dense layers that use ReLU as activation also have Dropout set to 0.5. The idea behind the model was to make a simple model that was able to achieve 90% or higher accuracy on MNIST[2]. MNIST is a very simple dataset, so the first network that was tested gave us the desired result with a small margin.

3.4 Application

3.4.1 Containernet Topology

Using Containernet, two separate applications for the hosts were created. One for the client and another for the server (Figure 7). The topology of hosts was created with two clients and one server, with switches connecting all of them to allow for message exchanges.

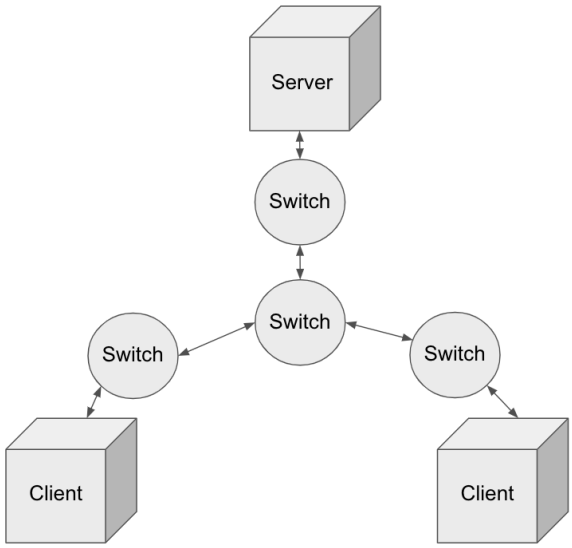

Figure 7: Container net topology diagram

For each link in the topology, a 100ms delay was attributed to emulate network latency.

In the context of a Vertical Federated Learning application, the two clients will take part as the two data holders and the server will be the partially trusted third-party.

We considered two host types of applications using python, one WebSocket server that will act as the trusted third-party, and one WebSocket client that will act as the two data holders. [1]

For WebSocket message exchanges, the WebSocket[5] python library was used, alongside asyncio[6], for allowing asynchronous handling of requests and responses from the applications.

All exchanged messages between the applications are encrypted using homomorphic encryption to allow for model training without exposing the data. For this task, the PyfHel[7] library was used.

The messages exchanged contain a serialized representation of the PyfHel encrypted objects, to accomplish that, the library jsonpickle[8] is used to serialize the objects and send them via JSON payloads.

Both the client hosts keep and train a machine learning model using the library PyTorch[9].

3.4.2 The Application Life Cycle

The application goes through multiple states during the training process, everything is based on a request/response approach to each operation to be executed. Mostly the server sends operation requests to the application hosts, which in turn sends their responses with the results of the operation.

Startup Initially, it is expected that only the server is active and listening. The server holds parameters that initially the applications are unaware of, and they must be transmitted so that we can guarantee the consistency of the cluster.

The server holds the *learning rate* that will be used during training, the *seed* that will be used to fetch the data in the same order in both clients, and the *batch size* for running the epochs.

The server expects two clients to connect before it starts the training process, while that condition is not met, it keeps waiting. If a client connects to the server, it sends the parameters to the newly connected host and waits for confirmation that the host is ready to start the training process.

Training After the two clients are connected and ready, the training process starts.

Epoch Initialization At the start of a new epoch, the server sends a request for both clients to initialize a new epoch, which in turn, the clients reset their dataloader iterators. The server waits for both clients to respond with a confirmation before proceeding.

Batch Step After initializing an epoch, the server proceeds with the training. Requests to run the batches with the configured batch size are sent to the clients.

First, the request is sent to the client that does not hold the labels of the data (we will call it the **secondary client**), which in turn, runs the batch data through its local model, and responds with its loss and gradient values.

Then the server sends another batch training request to the client that holds the labels (we will call it the **primary client**) with the data received from the secondary client. With that, the primary client will run the data through its model and respond to the server with the gradients that will be used to train both models and the model loss.

[5] `https://websockets.readthedocs.io/en/stable/intro.html`
[6] `https://docs.python.org/3/library/asyncio.html`
[7] `https://pyfhel.readthedocs.io/en/latest/`
[8] `https://jsonpickle.github.io/`
[9] `https://pytorch.org/`

With the gradient values, the server sends a request to execute a backpropagation operation to both clients, which updates their models and responds confirms the operation to the server.

That process will repeat until both clients have gone through their entire dataset. When that happens, the secondary client will notify the server that the epoch has ended with a flag in the batch step operation response.

Epoch End After the flag indicating the end of an epoch is received, the server will start another epoch, starting with the initialization of both clients again, and repeating the batch step operations. This will loop until the determined amount of epochs is met.

After the training operation ends, both clients hold the trained model without any of their data being exposed to each other.

4 Results

The model trained with VFL took 271.1 hours to train 9 epochs, with the epochs ranging from 29h to 32h and a standard deviation of 1h. The same model trained with a traditional centralized approach takes only 2 minutes.

Hardware used: i7 4770k, 32GB RAM.

The charts below contrast the results from the Vertical Federated Learning with a centralized model.

In Figure 8 we show the accuracy of both models. It is possible to see that the centralized model showed a quicker accuracy increase in the first few epochs when compared to VFL. The final model accuracy was 91.31% for VFL and 91.48% for the centralized. We found that this accuracy gap is negligible, given the small number of epochs and that this could be obtained by simply running the algorithm with a different seed.

The loss charts are also very much alike, excluding the "Loss per epoch" (Figure 11) charts because of the Dropout layers used. The training loss for the final epoch was 0.21420 for VFL and 0.36973 for the centralized. The validation loss (Figure 12) for the final epoch was 0.16178 for VFL and 0.00258 for the centralized. This difference in the validation loss was due to the fact that only one side of the network was used to train the VFL. The "Loss per batch (1 in 10)" (Figure 10) is a smoothed version of "Loss per batch" (Figure 9).

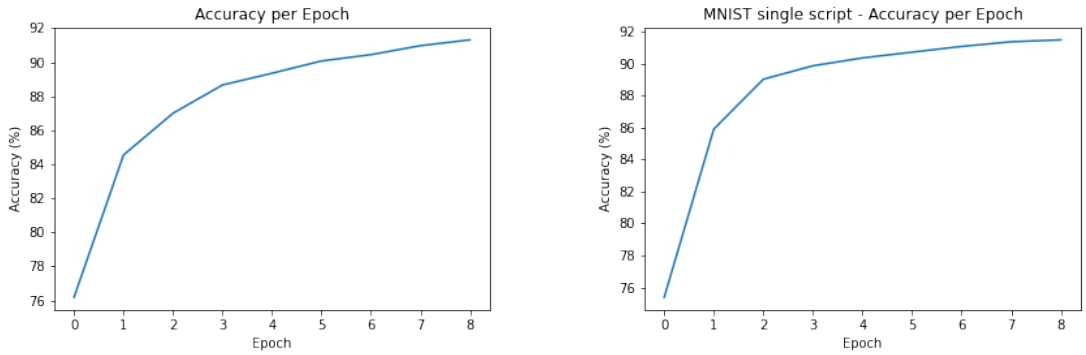

Figure 8: Charts comparing the accuracy of the networks trained by VFL (on the left) and centralized (on the right).

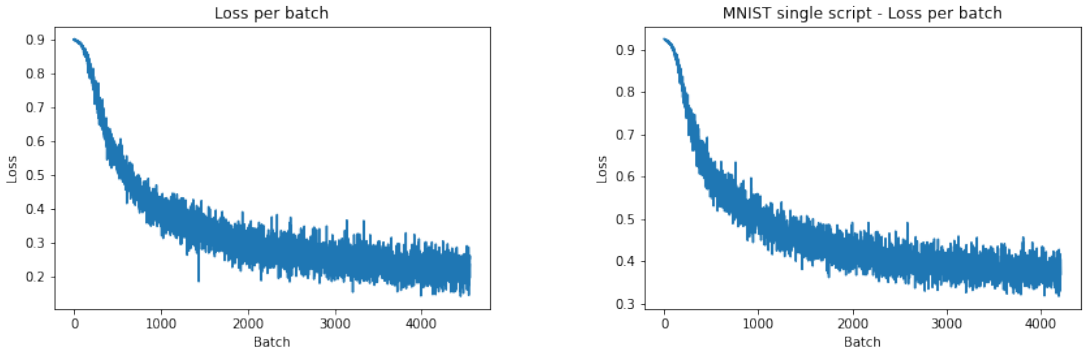

Figure 9: Charts comparing the loss per batch of the networks trained by VFL (on the left) and centralized (on the right).

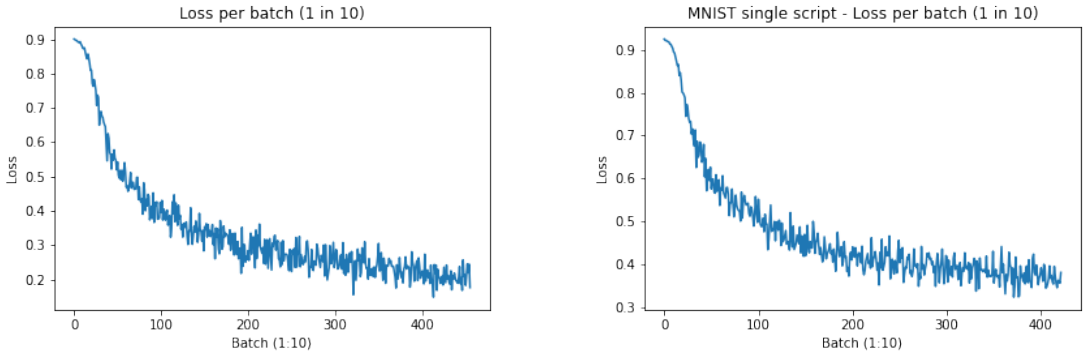

Figure 10: Charts comparing the loss every 10 batches of the networks trained by VFL (on the left) and centralized (on the right).

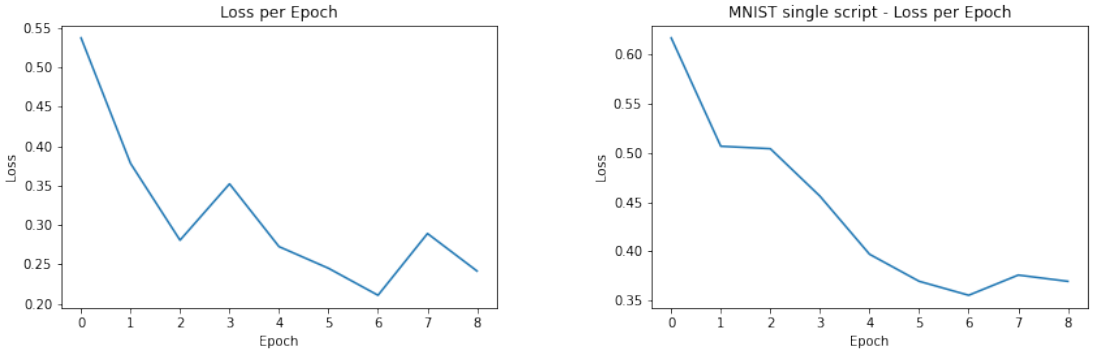

Figure 11: Charts comparing the losses of the networks trained by VFL (on the left) and centralized (on the right).

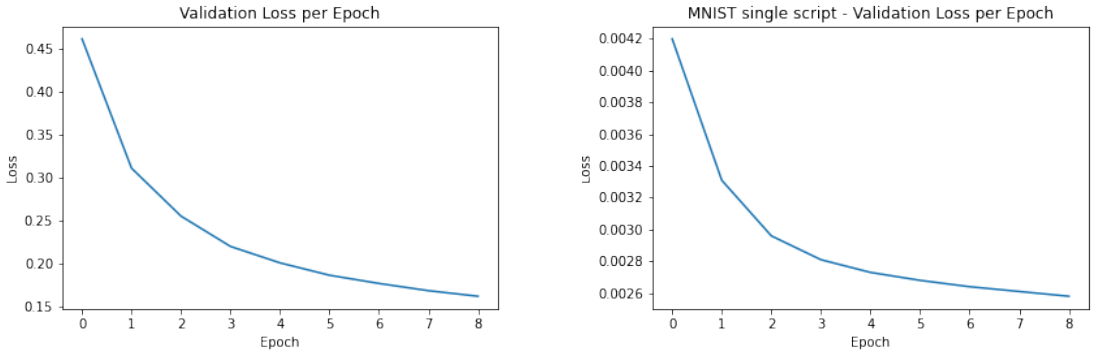

Figure 12: Charts comparing the validation loss per epoch of the networks trained by VFL (on the left) and centralized (on the right).

5 Conclusion

We showed that Vertical Federated learning is a method for creating collaborative models without sharing data, giving comparable results to a traditionally trained machine learning model. However, we could not demonstrate that it is yet practical to use VFL in production for more interesting models due to the big overhead that comes with it. It took more than 8000 times more time to train the VFL model than it takes to train a model locally.

We found that the usage of homomorphic encryption causes a very significant impact on the training performance. We believe that with improvements in the speed of the encryption and encrypted operations, Vertical Federated Learning could become practical for bigger and deeper models.

Further improvements to this work would be to compare quantitatively the overhead caused by the network transmission and the encryption, compare the centralized training to the VFL without the cryptography, test with different data distributions and with more datasets.

It is possible to use the provided application to train any kind of machine learning model using Vertical Federated Learning, with or without homomorphic encryption. Also, it can be used as a reference for PyfHel objects serialization and transmission, which is lacking through official means.

References

[1] Application Source Code `https://github.com/Trindad/containernet/tree/applications/applications`

[2] "THE MNIST DATABASE of handwritten digits". Yann LeCun, Courant Institute, NYU Corinna Cortes, Google Labs, New York Christopher J.C. Burges, Microsoft Research, Redmond.

[3] Communication-Efficient Learning of Deep Networks from Decentralized Data, H. Brendan McMahan and al. 2017

[4] Federated Learning for Mobile Keyboard Prediction, Andrew Hard and Chloé M Kiddon and Daniel Ramage and Francoise Beaufays and Hubert Eichner and Kanishka Rao and Rajiv Mathews and Sean Augenstein, 2018

[5] Federated Machine Learning: Concept and Applications, Qiang Yang and Yang Liu and Tianjian Chen and Yongxin Tong, 2019

[6] Asymmetrical Vertical Federated Learning, Yang Liu and Xiong Zhang and Libin Wang, 2020

[7] VAFL: a Method of Vertical Asynchronous Federated Learning, Tianyi Chen and Xiao Jin and Yuejiao Sun and Wotao Yin, 2020

[8] Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption, Stephen Hardy and Wilko Henecka and Hamish Ivey-Law and Richard Nock and Giorgio Patrini and Guillaume Smith and Brian Thorne, 2017

[9] A Guide to Fully Homomorphic Encryption, Frederik Armknecht and Colin Boyd and Christopher Carr and Kristian Gjøsteen and Angela Jäschke and Christian A. Reuter and Martin Strand, 2015

[10] DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, Victor SANH, Lysandre DEBUT, Julien CHAUMOND, Thomas WOLF, 2019.