|  |
| --- |
| Federated Learning Through Distributed Computing  WQD7008 Group Project |
| Team Members:  **Resmy Vijaykumar (WQD190042)**  **Liu Hong Yang (WQD190005)**  **Mohamed Fathi (WQD180081)**  **Sritharan Sivaguru (WQD180086)**  **Abu Zafar Mohammad Saleh (WQD180084)** |



Contents

[Introduction 3](#_Toc27346287)

[Definition of federated learning 3](#_Toc27346288)

[How federated learning works 3](#_Toc27346289)

[Advantages of federated learning 4](#_Toc27346290)

[Federated Learning Applications 5](#_Toc27346291)

[Scenarios 5](#_Toc27346292)

[Project Methodology 8](#_Toc27346293)

[Objective 8](#_Toc27346294)

[Data Definition: 8](#_Toc27346295)

[Project Components 9](#_Toc27346296)

[Prototyping 11](#_Toc27346297)

[Results 15](#_Toc27346298)

[Final Implementation 16](#_Toc27346299)

[Conclusion 18](#_Toc27346300)

[References 19](#_Toc27346301)

# **Introduction**

## **Definition of federated learning**

Mobile computing devices have seen a rapid increase in their computational power as well as storage capacity. Aided by this increased computational power and abundance of data, as well as due to privacy and security concerns, there is a growing trend towards training machine learning models over networks of such devices using only local training data.

The field of Federated learning initiated in McMahan et al. (2017) [1] and Konecn y et al. (2016) [7] considers the problem of learning using a centralized model based on private training data of a large number of users. More specifically, this framework is characterized by a huge number of decentralized users who are connected to a centralized server. The different users generate possibly non-IID (Independent and Identically Distributed) data and furthermore it is assumed that communications between the users and the central server incur large costs.

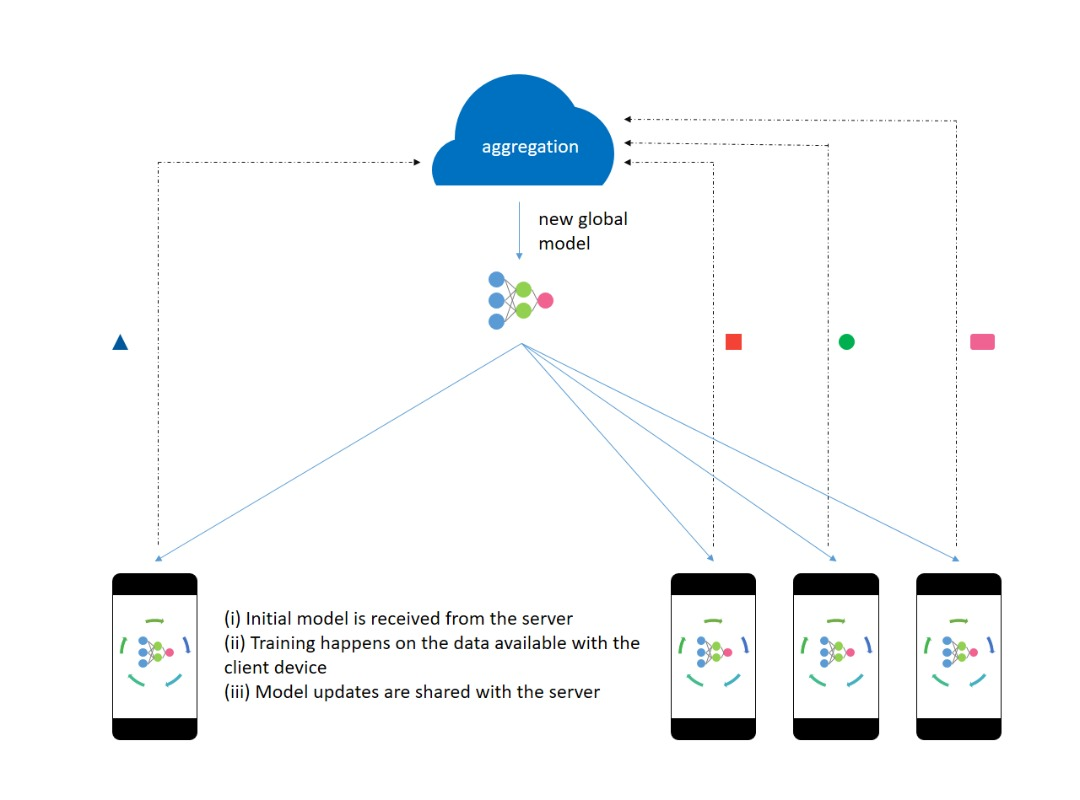
McMahan et al. (2017) [1] proposed the federated optimization algorithm in which the central server randomly selects a fraction of the users in each round, shares the current global model with them, and then averages the updated models sent back to the server by the selected users.

Federated Learning enables mobile phones to collaboratively learn a shared prediction model while keeping all the training data on device, decoupling the ability to do machine learning from the need to store the data in the cloud.

## **How federated learning works**

Federated Learning is a machine learning setting where the goal is to train a high-quality centralized model with training data distributed over a large number of clients each with unreliable and relatively slow network connections.

It works like this: your device downloads the current model, improves it by learning from data on your phone, and then summarizes the changes as a small focused update. Only this update to the model is sent to the cloud, using encrypted communication, where it is immediately averaged with other user updates to improve the shared model. All the training data remains on your device, and no individual updates are stored in the cloud.



**Figure 1.1: Typical federated learning setup**

## **Advantages of federated learning**

Federated Learning allows for

* smarter models,
* lower latency, and
* less power consumption, all while ensuring privacy.

And this approach has another immediate benefit: in addition to providing an update to the shared model, the improved model on your phone can also be used immediately, powering experiences personalized by the way you use your phone.

Federated learning enables data scientists to create AI without compromising user’s confidentiality. This method is set to disrupt the centralized AI paradigm, in which better algorithm always comes at the cost of collecting more and more personal data. Thus, federated learning allows powerful network effects in industries where data cannot be transferred to third parties due to confidentiality reasons (health records, banking transactions, etc.).

Personal data never leaves the user’s device, only updates made to the model are transferred. This data could possibly be encrypted, making it impossible for anyone to intercept the data and retro engineer it.

The updates are lighter than the original users’ data, consequently the overall workload needed is lower in federated learning than in cloud-based architectures or in edge computing, which makes it cheaper and more convenient.

The model is located in the user’s device, allowing for real time inferences with no latency problems

Federated learning empowers sectors where data cannot be transferred to third parties for confidentiality reasons with data network effects.

## **Federated Learning Applications**

Federated Learning is expected to be used in a lot of different applications. Here we list some as examples:

**AI on Mobile Devices**

With the Gboard (a virtual keyboard on Android phones), a case for Google as a first example, we expect federated learning to be widely used for AI on mobile devices including intelligent image processing for cameras, better voice recognition and NLU (Natural Language Understanding) for virtual assistants, improved recommendation system for advertisements etc. Most of the existing pipelines of collecting mobile data into a centralized server for analysis can be developed with federated learning.

**Healthcare**

Health data is extremely private for patients, and people are looking into using federated learning to train machine learning/deep learning models for healthcare applications while protecting the privacy of patients.

**Finance**

Federated learning can be expected to also apply to a wide range of financial use cases. For example, for anti-fraud across different banks, for risk modelling and pricing with banks and insurance companies collaborating with 3rd parties like internet companies that have a lot more customer data.

The above use cases would be further elaborated in the next section.

# **Scenarios**

Federated Learning is the new machine learning algorithm that enables distributed clients to handle decentralized data without controlled by the central server. This means that the processes happened on data owners (e.g. mobiles, computers, wireless devices) which store data and train data using machine learning algorithms locally. With the increasing computation performance of edge devices, the machine learning model has been trained on distributed devices. In federated learning, the new paradigm is different from the traditional machine learning environment which requires people collecting data stored on the central server and then use it to build the ideal model. It has changed to compute the user data set on edge devices and send the update to the central server. The new scenario of federated learning has been deployed on some applications [1].

A close up of a map

Description automatically generated

**Figure 2.1: next-word prediction on mobiles**

For example, because of the information protection of average users, some people are less likely to share their private information for companies to analyze their actions. One solution for these companies to continue doing the next-word prediction would be applying federated learning on edge mobiles and the selection of mobile phones would also be random. Furthermore, edge mobiles would perform the training on their local data using the machine learning algorithm and only the results would be sent to the central server for an update. After that, the central cloud server would aggregate and train these results at using the global model and sends the improved model to clients. The process may last for several iterations until the prediction function has been completed.

The example in figure 2.1 shows that in federated learning, and personal information on mobiles would not be collected or even leak, the company can also use the algorithm to finish the text- prediction applications on mobile phones [2].

The architecture of figure 2.2 has illustrated the horizontal federated learning system, the system consists of one server and several individual clients.

In the system, edge clients have similar data functions and databases. They collaborate on training data locally. This solution ensures data security without exposure to external devices [3].Specifically, the procedures include four steps: edge nodes learn the machine learning algorithm locally and send anonymous data to central Server A. The Server aggregates the results from clients to train a global model. The new aggregated model is sent back to clients. Finally, the clients would update its model, and the entire process would repeat iteratively until the model hits its maximum training rounds [4].

A close up of a map

Description automatically generated

**Figure 2.2: The architecture of a federated learning system**

The architecture above is classified as horizontal federated learning. In the scenario, data sets own the same data features, but the number of samples is different. Google uses the transfer learning model for training locally-stored data on Gboard to predict emoji. This approach helps protect users' privacy information without data leakage [5]. Horizontal federated learning has been utilized in the anti-laundering field. The Webank's AI department proposes the horizontal federated learning model to enable multiple organizations building an anti-laundering model called homogeneous logistic regression (Homo-LR) [6].

A screenshot of a cell phone

Description automatically generated

**Figure 2.3: Horizontal Federated Learning**

After illustrating the principles of horizontal federated learning, we decide to apply the algorithm on text messages to detect spam messages. In the submitted project, we will use compute nodes to simplify the system between the central server/gateway and remote nodes. Furthermore, they would communicate with each other through the Internet. The scenario shows what it would be like for a telco to train a deep learning model to detect spam without having access to their user’s text messages. Each mobile user would act as a slave node, with the telco acting as the gateway or master server that maintains the deep learning model. The master server would collect updates from multiple slave nodes on what is and isn’t a spam message to train a global model for detecting spam messages. The text message collection data set in each node/user will remain on the user’s device to protect the user’s privacy.

# **Project Methodology**

## **Objective**

In this project we will show how we can use Federated Learning with PySyft, the extension of PyTorch for a classification task using deep learning. The objective here is to simulate remote VMs in a cloud environment to perform federated learning, where each node has a similar number of labelled datapoints (text messages labelled as spam or not), to train a neural network model without each node revealing its data to other nodes or the gateway. As described in the previous section, this is to simulate a use case where a telco could train a deep learning model that could be used by all users in their network, without any user revealing the content of their text messages to the telco or other users. In a real scenario, the model will be trained by what the user identifies as spam. However, in our simulation, the text messages that are spam would be pre-determined as part of the data set to simplify the training.

## **Data Definition:**

The text messages used for testing has been collected from free or free for research sources in the Internet:

* A collection of 425 SMS spam messages was manually extracted from the Grumbletext Web site. This is a UK forum in which cell phone users make public claims about SMS spam messages, most of them without reporting the very spam message received. The identification of the text of spam messages in the claims is a very hard and time-consuming task, and it involved carefully scanning hundreds of web pages. The Grumbletext Web site is: [[Web Link]](http://www.grumbletext.co.uk/).
* A subset of 3,375 SMS randomly chosen ham messages of the NUS SMS Corpus (NSC), which is a dataset of about 10,000 legitimate messages collected for research at the Department of Computer Science at the National University of Singapore. The messages largely originate from Singaporeans and mostly from students attending the University. These messages were collected from volunteers who were made aware that their contributions were going to be made publicly available. The NUS SMS Corpus is available at: [[Web Link]](http://www.comp.nus.edu.sg/~rpnlpir/downloads/corpora/smsCorpus/).  
  A list of 450 SMS ham messages collected from Caroline Tag's PhD Thesis available at [[Web Link]](http://etheses.bham.ac.uk/253/1/Tagg09PhD.pdf).
* Finally, we have incorporated the SMS Spam Corpus v.0.1 Big. It has 1,002 SMS ham messages and 322 spam messages and it is public available at: [[Web Link]](http://www.esp.uem.es/jmgomez/smsspamcorpus/)

**Attribute Information:**

The collection is composed by just one text file, where each line has the correct class followed by the raw message. Some examples bellow:

|  |  |
| --- | --- |
| Type | Text |
| ham | What you doing?how are you? |
| ham | Ok lar... Joking wif u oni... |
| ham | dun say so early hor... U c already then say... |
| ham | MY NO. IN LUTON 0125698789 RING ME IF UR AROUND! H\* |
| ham | Siva is in hostel aha:-. |
| ham | Cos i was out shopping wif darren jus now n i called him 2 ask wat present he wan lor. Then he started guessing who i was wif n he finally guessed darren lor. |
| spam | FreeMsg: Txt: CALL to No: 86888 & claim your reward of 3 hours talk time to use from your phone now! ubscribe6GBP/ mnth inc 3hrs 16 stop?txtStop |
| spam | Sunshine Quiz! Win a super Sony DVD recorder if you canname the capital of Australia? Text MQUIZ to 82277. B |
| spam | URGENT! Your Mobile No 07808726822 was awarded a L2,000 Bonus Caller Prize on 02/09/03! This is our 2nd attempt to contact YOU! Call 0871-872-9758 BOX95QU |

Note: the messages are not chronologically sorted.

**Python & ML Libraries used**

PySyft, an open-source library built for Federate Learning and Privacy Preserving. PySyft allows its users to perform private and secure Deep Learning (DL). It is built as an extension of some DL libraries, such as PyTorch, Keras and Tensorflow.

Other libraries such as: Numpy, panda, sickit learn, PyGrid are used for this project.

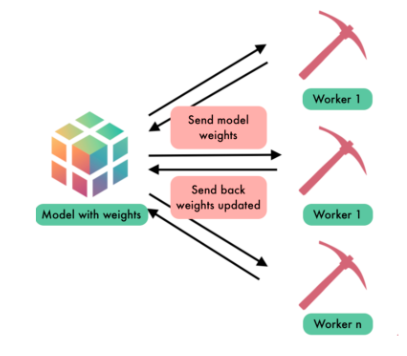
# **Project Components**

The final implementation involves the following components:

1. Client
2. Gateway
3. Worker Nodes

We will start with a prototype to validate our federated learning implementation using virtual workers. This would make it easy to first test our implementation before proceeding to implement it into separate VMs performing the function of gateway and worker nodes.

Figure 3.1 depicts how our implementation would eventually look like, with the master node iterating the over the model, while the workers train the model by updating the model weights. After the desired accuracy is reached (typically over 15 epochs), the training will end and the results compared.



**Figure 3.1: Typical master (gateway) with worker nodes**

To begin, we need to pre-process the data set so that it can be loaded by workers in a format that could be used to train the federated learning model.

The dataset that was discussed earlier is downloaded and processed to convert the data from textual form into numerical data. This is required so that we can utilize a GRU (gated recurrent unit) RNN (recurrent neural network) to train the learning model for our federated learning implementation.

The dataset (spamsample.csv) will be broken down into to inputs using the pre-processing script (data\_prep.py):

1. msgtexts.npy – containing the messages converted into numerical data.
2. msgtypes.npy – containing the message types, whether the message is spam or not.

The values within the msgtypes.npy dataset corresponds to an NumPy array of 30 tokens generated from each text message padded at the left or truncated at right, to obtain a fixed size.

The msgtypes.npy dataset would only have a value of 1 for spam and 0 for non-spam messages.

Here is a textual sample of the data after pre-processing:

msgtexts:

0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 5.491000000000000000e+03 7.240000000000000000e+03 1.268000000000000000e+03 8.358000000000000000e+03 4.350000000000000000e+03 7.508000000000000000e+03 8.248000000000000000e+03 2.011000000000000000e+03 4.840000000000000000e+03 6.833000000000000000e+03 4.368000000000000000e+03 1.909000000000000000e+03 5.181000000000000000e+03 2.205000000000000000e+03 2.817000000000000000e+03 3.936000000000000000e+03 3.665000000000000000e+03 6.511000000000000000e+03 5.168000000000000000e+03 2.808000000000000000e+03

0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 0.000000000000000000e+00 6.442000000000000000e+03 1.850000000000000000e+03 1.788000000000000000e+03 3.501000000000000000e+03 7.000000000000000000e+01 2.535000000000000000e+03

…

msgtypes:

0.000000000000000000e+00

0.000000000000000000e+00

1.000000000000000000e+00

0.000000000000000000e+00

0.000000000000000000e+00

1.000000000000000000e+00

0.000000000000000000e+00

0.000000000000000000e+00

1.000000000000000000e+00

1.000000000000000000e+00

…

The current implementation of PySyft does not support the RNN modules of PyTorch directly yet, we will need to create a simplified version that could be used to train our model. Details on the GRU RNN that is being adapted is available here: (<https://pytorch.org/docs/stable/nn.html#gru>). The custom GRU RNN is written into a script (GRU\_RNN.py) that will be imported into the client later when invoking the gateway to perform the federated learning.

## **Prototyping**

Once we have prepared the data and the model we would like to use. We proceed to write a prototype that we can use to test our data and implementation. For this purpose, we use Google Colab Notebook with the Python 3 runtime and GPU enabled. The use of a TPU is possible, but we had not been successful in getting the setup using a TPU to work.

First, we load PySyft and PyTorch:

!pip install syft torch==1.3.0 torchvision==0.4.1 -f https://download.pytorch.org/whl/torch\_stable.html

Because PyTorch does not yet support TensorFlow 2, we must limit the TensorFlow backend used by Google Colab to version 1.x as follows:

%tensorflow\_version 1.x

We then proceed to check for CUDA support that will be used by PyTorch and assign it to a variable for reference later:

import torch as tc

if tc.cuda.is\_available():

  print(tc.cuda.get\_device\_name(0))

  print(tc.cuda.device\_count())

  tc.set\_default\_tensor\_type(tc.cuda.FloatTensor)

  cuda\_device = tc.device('cuda:0')

*Note*: If a CUDA device is not detected, please check your runtime settings. The preceding codes will fail to execute if there is no GPU available.

We import the PySyft library at this stage and assign the PyTorch tensor that we imported into the notebook earlier. This allows PySft to interface with PyTorch as it’s backend. We also create 2 worker nodes (Worker1 and Worker2) that will be used for our training later. To ease setup, we will create these workers as virtual nodes that are part of the same python runtime:

import syft as sf

hook = sf.TorchHook(tc)

w1 = sf.VirtualWorker(hook, id="worker1")

w2 = sf.VirtualWorker(hook, id="worker2")

We upload the msgtypes and msgtext pre-processed data into notebook to be used for training:

from google.colab import files

upl = files.upload()

import io

import numpy as np

After successful loading of the data into the notebook, we import it into the following variables, texts for the text messages and types for the message types:

texts = np.load(io.BytesIO(upl["msgtexts.npy"]))

texts = tc.tensor(texts).to(cuda\_device, tc.long)

types = np.load(io.BytesIO(upl["msgtypes.npy"]))

types = tc.tensor(types).to(cuda\_device, tc.long)

We split the data into 80% for training and 20% for testing, then spit it into 2 halves for Worker1 and Worker2:

# Split training and test data

test\_pct = 0.2

train\_types = types[:-int(len(types)\*test\_pct)]

train\_texts = texts[:-int(len(types)\*test\_pct)]

test\_types = types[-int(len(types)\*test\_pct):]

test\_texts = texts[-int(len(types)\*test\_pct):]

# Dataset split (one half for w1, other half for w2)

train\_idx = int(len(train\_types)/2)

test\_idx = int(len(test\_types)/2)

We then send the data sets (both test and training datasets) to the workers for processing. Note that the data can’t be processed in the notebook directly, as that would not be federated learning. It has to be done on separate (worker) nodes and not visible to the master (notebook):

w1\_train\_dataset = sf.BaseDataset(train\_texts[:train\_idx],

                                  train\_types[:train\_idx]).send(w1)

w2\_train\_dataset = sf.BaseDataset(train\_texts[train\_idx:],

                                    train\_types[train\_idx:]).send(w2)

w1\_test\_dataset = sf.BaseDataset(test\_texts[:test\_idx],

                                  test\_types[:test\_idx]).send(w1)

w2\_test\_dataset = sf.BaseDataset(test\_texts[test\_idx:],

                                  test\_types[test\_idx:]).send(w2)

We define a global value for the batch processing size that will be performed by each worker:

BATCH\_SIZE = 32

We specify that that the datasets that were sent to the workers are to be processed in a federated manner:

federated\_train\_dataset = sf.FederatedDataset([w1\_train\_dataset, w2\_train\_dataset])

federated\_test\_dataset = sf.FederatedDataset([w1\_test\_dataset, w2\_test\_dataset])

federated\_train\_loader = sf.FederatedDataLoader(federated\_train\_dataset,

                                                shuffle=True, batch\_size=BATCH\_SIZE)

federated\_test\_loader = sf.FederatedDataLoader(federated\_test\_dataset,

                                               shuffle=False, batch\_size=BATCH\_SIZE)

We define the number of training epochs that we want to perform using the GRU model we have defined earlier:

EPOCHS = 15

CLIP = 5

lr = 0.1

# Model parameters

VOCAB\_SIZE = int(texts.max()) + 1

EMBEDDING\_DIM = 50

HIDDEN\_DIM = 10

DROPOUT = 0.2

# Initiating the model

model = GRU(vocab\_size=VOCAB\_SIZE, hidden\_dim=HIDDEN\_DIM, embedding\_dim=EMBEDDING\_DIM, dropout=DROPOUT)

We perform the training and test of the model defined earlier:

from sklearn.metrics import roc\_auc\_score

# Defining the loss and optimizer (which will be discussed later)

criterion = nn.BCELoss()

optimizer = tc.optim.SGD(model.parameters(), lr=lr)

for e in range(EPOCHS):

    # To track the amount of loss

    losses = []

    # Batch processing loop for training

    for texts, types in federated\_train\_loader:

        # Location of current batch

        worker = texts.location

        # Initialize hidden state and send it to the worker

        h = tc.Tensor(tc.zeros((BATCH\_SIZE, HIDDEN\_DIM))).send(worker)

        # Send the model to the current worker

        model.send(worker)

        # Accumulated gradients to zero before optimization step

        optimizer.zero\_grad()

        # Output from the model

        output, \_ = model(texts, h)

        # Calculate the loss and perform backprop

        loss = criterion(output.squeeze(), types.float())

        loss.backward()

        # Clipping the gradient to avoid explosion

        nn.utils.clip\_grad\_norm\_(model.parameters(), CLIP)

        # Backpropagation step

        optimizer.step()

        # Get the model back to the master

        model.get()

        losses.append(loss.get())

    # Evaluate the model

    model.eval()

    with tc.no\_grad():

        test\_preds = []

        test\_types\_list = []

        eval\_losses = []

        for texts, types in federated\_test\_loader:

            # Location of current batch

            worker = texts.location

            # Initialize hidden state and send it to worker

            h = tc.Tensor(tc.zeros((BATCH\_SIZE, HIDDEN\_DIM))).send(worker)

            # Send the model to the worker

            model.send(worker)

            output, \_ = model(texts, h)

            loss = criterion(output.squeeze(), types.float())

            eval\_losses.append(loss.get())

            preds = output.squeeze().get()

            test\_preds += list(preds.cpu().numpy())

            test\_types\_list += list(types.get().cpu().numpy().astype(int))

            # Get the model back to the master

            model.get()

        score = roc\_auc\_score(test\_types\_list, test\_preds)

    print("Epoch {}/{}...  \

    AUC: {:.3%}...  \

    Training loss: {:.5f}...  \

    Validation loss: {:.5f}".format(e+1, EPOCHS, score, sum(losses)/len(losses), sum(eval\_losses)/len(eval\_losses)))

    model.train()

The output from the training is explained in the next section.

# **Results**

After approximately 15 minutes of training and evaluation, we receive the following output:

Epoch 1/15... AUC: 71.638%... Training loss: 0.43475... Validation loss: 0.36051

Epoch 2/15... AUC: 80.175%... Training loss: 0.36033... Validation loss: 0.32753

Epoch 3/15... AUC: 86.964%... Training loss: 0.32431... Validation loss: 0.28528

Epoch 4/15... AUC: 92.736%... Training loss: 0.27732... Validation loss: 0.23043

Epoch 5/15... AUC: 95.891%... Training loss: 0.22327... Validation loss: 0.17480

Epoch 6/15... AUC: 97.087%... Training loss: 0.17901... Validation loss: 0.13715

Epoch 7/15... AUC: 97.577%... Training loss: 0.14626... Validation loss: 0.11782

Epoch 8/15... AUC: 97.969%... Training loss: 0.12828... Validation loss: 0.10451

Epoch 9/15... AUC: 98.310%... Training loss: 0.11127... Validation loss: 0.09531

Epoch 10/15... AUC: 98.532%... Training loss: 0.09696... Validation loss: 0.09265

Epoch 11/15... AUC: 98.777%... Training loss: 0.08692... Validation loss: 0.08423

Epoch 12/15... AUC: 98.730%... Training loss: 0.08059... Validation loss: 0.08605

Epoch 13/15... AUC: 98.939%... Training loss: 0.07064... Validation loss: 0.07467

Epoch 14/15... AUC: 98.811%... Training loss: 0.06795... Validation loss: 0.08034

Epoch 15/15... AUC: 98.735%... Training loss: 0.06020... Validation loss: 0.07952

To help understand the above output, we need to understand the following:

1. Stochastic gradient descent (SGD) is an iterative learning algorithm that uses a training dataset to update a model. (The descent was limited using the CLIP variable).
2. The BATCH\_SIZE is a parameter of gradient descent that controls the number of training samples to work through before the model’s internal parameters are updated.
3. The number of EPOCHS is a parameter of the gradient descent that controls the number of complete passes through the training dataset.

Stochastic Gradient Descent, or SGD (as was referenced in tc.optim.SGD), is an optimization algorithm used to train neural network machine learning algorithms, in this case it is the GRU model that was used in the prototype. The objective of the algorithm is to find a set of internal model parameters that perform well against the performance measure, training and validation loss. The algorithm is iterative as it occurs over multiple steps, each step slightly improving the model parameters, as can be seen from the output. Each step involves using the model to make predictions on the provided samples, comparing the predictions to the expected outcomes (test data), calculating the error, and using the error to update the internal model parameters.

The BATCH\_SIZE is a parameter that defines the number of samples to work through before updating the internal model parameters. It is a for-loop iterating over the samples and making predictions. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down the error gradient.

The number of EPOCHS is a parameter that defines the number times that the learning algorithm will work through the entire training dataset. It is a for-loop over the number of EPOCHS, where each loop proceeds over the training dataset. Within this for-loop is another nested for-loop that iterates over each batch of samples, where one batch has the specified BATCH\_SIZE number of samples.

AUC (Area Under the Curve) is a performance measurement for a classification problem at various thresholds. Technically, AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. The higher the AUC score, better the model is at predicting 0 as 0 (not spam) and 1 as 1 (spam). For this dataset, the higher the AUC, the better the model is at distinguishing between text messages that are spam and those that are not spam.

The derived accuracy of 98.7% (AUC) is sufficient and we can now proceed to redefine our prototype code to meet our objective of running on actual VMs.

## **Final Implementation**

To split up the code and execute it on nodes, we will use PyGrid (<https://github.com/OpenMined/PyGrid>). It is a framework to extend PySyft into deployable tools that could be used in VMs and Docker containers.

*Note*: Due to the complexity of deploying to Google Compute Engine (that would be used in the actual demo of the implementation), we will provide simple instructions for deploying to any Linux host or VM in this report. We’ve also removed the need for GPUs to simplify the set up further.

**Requirements:**

* Git
* Python 3.7+ (installed on your PC and VMs)
* 3 Linux VMs/hosts (1 gateway, 2 worker nodes)
* 1 PC (to connect to gateway and nodes)

**Step 1:**

To begin, we will need to clone the git repository into each VM. We will need at least 3 VMs, one for the gateway, two for worker nodes, just like our prototype. To clone the repository execute the following command on each VM:

git clone <https://github.com/OpenMined/PyGrid.git>

**Step 2:**

We then enter the PyGrid directory on each VM and install the perquisites:

cd PyGrid

pip install -r requirements.txt

**Step 3:**

On the gateway VM, execute the following:

cd gateway

python gateway.py –start\_local\_db –port=8080

**Step 4:**

Ensure to take down the gate way IP address as you will now need to replace <gateway ip> with the actual IP of the VM.

On worker node 1, execute the following:

cd app/websocket/

python websocket\_app.py –start\_local\_db –id=worker1 –port=3001 –gateway\_url=http://<gateway IP>:8080

On worker node 2, execute the following:

cd app/websocket/

python websocket\_app.py –start\_local\_db –id=worker2 –port=3001 –gateway\_url=http://<gateway IP>:8080

**Step 5:**

Ensure to take down the worker node IP addresses as you will now need to replace the <worker1 IP> and <worker2 IP> with the actual IP of the VMs.

From the submitted project files for this report, execute the following script on your PC. Note that we only upload the data to the nodes, not the gateway:

python upload\_data.py -node1 <worker1 IP> -node2 <worker2 IP>

python train\_model.py -gw <gateway IP>

After approximately 20mins, you should receive similar output as shown by the prototype.

# **Conclusion**

In this project we’ve demonstrated that it is possible to train a model with decent accuracy on distributed nodes. The gateway node is not privy to the information stored by the worker nodes, but is able to train and test the neural network model through each node. The nodes were also able to train the model without either node knowing what data the other node has. This implementation serves as an example of a practical way to perform deep learning using distributed nodes in a federated learning setup, for the purpose of detecting spam text messages without revealing the data from users (nodes).

# **References**

[1] “Federated Learning: Collaborative Machine Learning without Centralized Training Data.” *Google AI Blog*, 6 Apr. 2017, <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>.

[2] Li, et al. “Federated Learning: Challenges, Methods, and Future Directions.” *ArXiv.org*, 21 Aug. 2019, <https://arxiv.org/abs/1908.07873>.

[3] Hartmann, et al. “Federated Learning for Ranking Browser History Suggestions.” *ArXiv.org*, 26 Nov. 2019, <https://arxiv.org/abs/1911.11807?context=stat.ML>.

[4] Mittal, Saransh. *Federated Learning*. Towards Data Science, 9 Oct. 2019, <https://towardsdatascience.com/federated-learning-3097547f8ca3>.

[5] “A Beginners Guide to Federated Learning.” *A Beginners Guide to Federated Learning - By Dr. Santanu Bhattacharya*, <https://hackernoon.com/a-beginners-guide-to-federated-learning-b29e29ba65cf>.

[6] *Multi-Objective Evolutionary Federated Learning - IEEE Journals & Magazine*, <https://ieeexplore.ieee.org/abstract/document/8744465>.

[7] Konečný, et al. “Federated Learning: Strategies for Improving Communication Efficiency.” *ArXiv.org*, 30 Oct. 2017, <https://arxiv.org/abs/1610.05492>.

[8] Bonawitz, et al. “Towards Federated Learning at Scale: System Design.” *ArXiv.org*, 22 Mar. 2019, <https://arxiv.org/abs/1902.01046>.

[9] “Towards Efficient and Privacy-Preserving Federated Deep Learning” – IEEE conference paper <https://ieeexplore.ieee.org/abstract/document/8761267>

[10] Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., ... & Van Overveldt, T. (2019). Towards federated learning at scale: System design. arXiv preprint arXiv:1902.01046.

[11] Li, T., Sahu, A. K., Talwalkar, A., & Smith, V. (2019). Federated learning: Challenges, methods, and future directions. arXiv preprint arXiv:1908.07873.

[12] Li, S., Cheng, Y., Liu, Y., Wang, W., & Chen, T. (2019). Abnormal Client Behavior Detection in Federated Learning. arXiv preprint arXiv:1910.09933.

[13] Yang, Q., Liu, Y., Chen, T., & Tong, Y. (2019). Federated machine learning: Concept and applications. ACM Transactions on Intelligent Systems and Technology (TIST), 10(2), 12.

[14] Ramaswamy, S., Mathews, R., Rao, K., & Beaufays, F. (2019). Federated learning for emoji prediction in a mobile keyboard. arXiv preprint arXiv:1906.04329.

[14] (2019). Retrieved 9 December 2019, from https://www.fedai.org/cases/utilization-of-fate-in-anti-money-laundering-through-multiple-banks/