Implementing a Network Optimized Federated Learning Method From the Ground up

Gustav Dettner Källander and Henning Norén

***Abstract*—This bachelor thesis presents the implementation of a simple fully connected neural network (FCNN) and federated neural network with stochastic quantization from scratch and compares their performance. Federated learning enables multiple parties to contribute to a machine learning model without sharing their sensitive data. The federated learning approach is becoming increasingly popular due to its ability to train models on decentralized data sources while maintaining privacy and security. Both the FCNN and federated network are trained and tested on the Modified National Institute of Standards and Technology (MNIST) database, the first one achieving around 90% accuracy after 50 epochs while the federated architecture only able to reach around 45% accuracy. This remains the same when data is quantized.**

***Sammanfattning*—Denna kandidatuppsats presenterar implementeringen av ett enkelt fullt anslutet neuralt nätverk (FCNN) och ett federerat neuralt nätverk med stokastisk kvantisering från grunden, samt jämför deras prestanda. Federativt lärande möjliggör att flera parter kan bidra till en maskininlärningsmodell utan att dela med sig av känsliga data. Federativt lärande har blivit allt mer populärt på grund av dess förmåga att träna modeller på decentraliserade datakällor samtidigt som integritet och säkerhet bevaras. Både FCNN och det federerade nätverket tränas och testas på Modified National Institute of Standards and Technology (MNIST) databas, där det första uppnår cirka 90% noggrannhet efter 50 epoker medan den federerade arkitekturen endast når cirka 45% noggrannhet. Detta förblir detsamma när data kvantiseras.**

***Index Terms*—Deep Neural Network, Machine Learning, Federated Learning, Gradient Descent, MNIST, Stochastic Quantization.**

Supervisors: Seyedsaeed Razavikia

TRITA number: TRITA-EECS-EX-2023:177

I. Introduction

The field of machine learning (ML) and neural networks is continuously advancing, offering numerous applications in areas like image recognition and autonomous driving. However, with recent developments in large language models there has been a growing concern for data privacy and how data is accessed [1]. This is not a trivial problem, better performance of neural networks is usually necessitated by access to better and larger data sets. Over the last decades, the world has witnessed an explosion of connected devices, ranging from personal smart devices, home appliances, industrial monitoring and more. This network of connected devices is widely known as an Internet of Things (IoT). The availability of such devices have enabled new possibilities for gathering data from various sources leading to a vast amount of data to be transported, stored and computed. This has given rise to distributed methods of ML, where data from several devices is composited into one large model However, this data gathering comes at a cost of sacrificing data privacy.

In this thesis, the problem of privacy regarding the mass collection of user data is tackled by implementing a simple federated neural network from scratch. Code for the network is showcased in Appendix A. The federated neural network constructs a local model in each users' personal device which are then transmitted to a main processing unit and composited into a main model. By making each personal device construct a local model, the users' data gets muddled into a large set of parameters. Converting these parameters back into user data is practically impossible. Through this method the owner of the main model has no way of accessing user data, which heightens their privacy. The network is implemented in code from scratch and tested on the MNIST data set which is then compared to the results in the following paper: "Federated Learning With Quantized Global Model Updates" [2]. To optimize network performance, following the paper, a stochastic method of compression is also implemented. By constructing a simple fully connected neural network (FCNN) we can compare how federated learning affects convergence. The following section presents the theory and background for conventional machine learning methods and federated learning. Section three delves deeper into specifically how our implementation is constructed and why that is.

II. Theory

A. Supervised Learning

This thesis deals with supervised learning algorithms which are used to classify unidentified input data given a set of known labeled data. The known labeled data is a set of tuples of size N as shown in Equation 1.

$$(x_1, y_1), (x_2, y_2), ..., (x_N, y_N), \tag{1}$$

where $x_i \in \mathbb{R}^d$ is the input vector of i-th example and $y_i \in \{1, 2, ...C\}$ is its corresponding label for C number of categories. Supervised learning algorithms then seek (learn) a function $g : X \to Y$ such that g(x) is considered a "good" predictor of the output value y. Specifically we want to learn a function g that takes an input vector of dimension d as shown in Equation 1 and returns a correct category C for that specific vector. The function g is an element of some set of possible functions G, call it the hypothesis space. This hypothesis space can be any arbitrary function space, however for our case in image classification the function g will be a conditional probability model, as seen in Equation 2.

$$g(x) = P(y|x). \tag{2}$$

Essentially meaning that given an image x, what is the probability that the category is y? A machine learning algorithm sets out to learn the function g such that it is able to, as accurately as possible, classify the categories. [3] [4]

B. Image Classification

Image classification is the process of assigning a specific class to a given image. Multi-class classification concerns the image classification problems where there exists at least two mutually exclusive categories. Taking a specific example: given handwritten images of digits 0-9, seek a classification function which classify the specified digit in the image. In the most simple case, we can solve this as a standard linear regression problem where the output is a linear function of the input. Let us consider a model where $\hat{y}$ is what our model predicts the class y should be. Treating this as linear regression, we can define the output as in Equation 3.

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + b. \tag{3}$$

We can think of $\boldsymbol{w}$ as a set of weights that simply put determines how each feature of the image predicts the models prediction. A positive weight increases the value of the prediction and a large weight in magnitude has a larger effect on the prediction. The term b is called a bias term and comes from the fact that the predicted output is "biased" toward b in the absence of input. To classify unknown input more accurately, one needs a way of measuring the performance of the model. This will be done by defining a loss function, an evaluation of a candidate prediction. Two common loss functions used, are the mean-square loss (MSE) or l_2 norm and cross-entropy loss. For a set of N predictions, we define the mean-square loss and cross-entropy loss in Equation 4.

$$\mathcal{L}(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2, \mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{N} y_i * log(\hat{y_i}) \tag{4}$$

MSE represents the squared distance from the models candidate prediction to the correct classification. What remains is to learn $\mathbf{w}$ and b such that more accurate predictions are made. Since we have a measure of how accurate the model is, we can view it as an optimization problem where our goal is to minimize the loss function by changing the weights and biases. A method of finding these parameters is by utilizing Stochastic Gradient Descent (SGD). Gradient descent (GD) can be calculated as shown in Equation 5.

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta \mathcal{L}(\hat{y}, y), \tag{5}$$

where θ is a parameter to be changed (weight or bias) and η a step-size. When dealing with large samples and networks however, it becomes exceedingly slow to compute the gradient for all samples. To combat this, we use SGD where we calculate the gradient on only a randomly selected subset of the training data, called a batch. One iteration through the training data is called an epoch. The model for seeking g, will consist of a FCNN as explained below. It will learn on a set of images with assigned labels and then will be tested on a set of test data of images to see how accurate it is. [5]

C. Fully-connected Neural Network

A FCNN is an acyclic graph of layered neurons where all neurons between adjacent layers are fully pairwise connected. A neuron takes as input a scalar x, multiplies it by a weight w and adds a bias term b. The resulting value is saved as $z = wx + b$ and the neuron output a is achieved by computing z through an activation function.

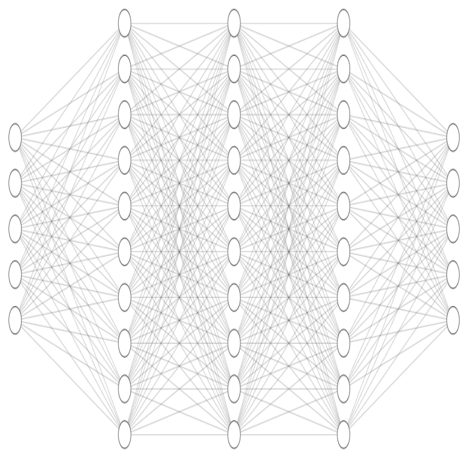

Fig. 1. Illustration of a fully connected neural network.

In Figure 1 each edge represents a weight and each node represents activation. A deep neural network consists of an input layer, some amount of hidden layers and an output layer. Each consisting of an arbitrary amount of neurons. Continuing with the example of image classification of 0-9 digits in the last section, the output layer would consist of 10 neurons corresponding to the 10 digits. The activation function is a nonlinear function of which there are many used, for example the Softmax and Sigmoid functions:

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \sigma(z) = \frac{1}{1+e^{-z}}. \tag{6}$$

Without activation functions, the neural network is simply a large amount of linear functions combined making it a very large linear regression model. When activation functions are added, non-linearity is introduced into the system meaning that more accurate predictions can be made on unknown complicated problems. Since we have an acyclic graph and considering that all neurons in adjacent layers are fully pairwise connected it is reasonable to represent the weights and biases as matrices and vectors. Let:

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,m} \\ w_{2,1} & w_{2,2} & \dots & w_{2,m} \\ \dots & \dots & \dots & \dots \\ w_{n,1} & w_{n,2} & \dots & w_{n,m} \end{pmatrix}, b = \begin{pmatrix} b_1 \\ . \\ . \\ . \\ b_n \end{pmatrix}, \tag{7}$$

where rows in W correspond to neurons in previous layer and columns correspond to neurons in next layer. Continuing, the output from each layer is propagated forward from first layer to the last. Can therefore represent the activations from one layer to the next as:

$$z^L = W^L a^{L-1} + b^L, a^L = \sigma(z^L), \tag{8}$$

with L referring to a given layer.

As mentioned previously, SGD or GD can be used to change the parameters of the model. However there are parameters whose values are set before training and remain unchanged. These are called hyperparameters which is explained next.

D. Hyperparameters and optimization

Hyperparameters play a critical role in the optimization process of machine learning algorithms. These parameters are set before the learning process begins and determine how the algorithm learns and generalizes from the training data. Unlike model parameters, which are learned during training, hyperparameters are not updated automatically but need to be specified by the user. Properly selecting and tuning hyperparameters can significantly impact the performance and generalization ability of a machine learning model.

In optimization, hyperparameters define the configuration of the learning algorithm. They control aspects such as the model's complexity, the learning rate, the number of iterations, and more. For instance, in a neural network, hyperparameters may include the number of hidden layers, the number of neurons in each layer, the learning rate and the batch size. The choice of hyperparameters can have a profound impact on the model's ability to converge and achieve good generalization. Where generalization essentially is a measure of how well the machine learning model predicts unseen data. To do this, one needs to test the network on a sample (of images in our case). How an image is input into the network and how to get a predicted output will be discussed next.

E. MNIST and testing the network

For the case of image classification, in order for the image to be input in the network a way of representing the image in a single vector needs to be obtained. This will be done by taking a black and white image and transforming it to a column vector where the amount corresponds to the amount of pixels in the image. Our thesis deals with image classification concerning images taken from MNIST database. MNIST database consists of 70000 handwritten images (greyscale) of digits 0 to 9. Example image is shown in Figure 2.

Knowing which digit is predicted through our model will be done by one-hot encoding. Meaning that label y representing the digit in the image, will be represented as a vector where all elements are zero except for the element corresponding to the labeled digit, which is one. Output prediction from the model will be of the same one-hot encoding format. For 10 neurons in the output layer, the predicted output digit corresponds to the highest maximum activation of the output layer. This ensures an unambiguous representation of the predicted digit.

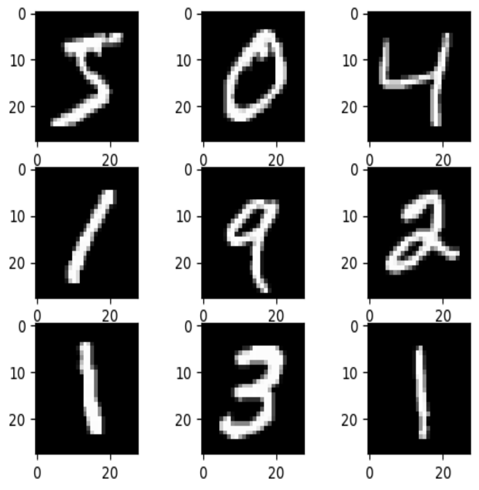

Fig. 2. Example images from MNIST data set.

F. Training

We have already defined a loss function which measures the accuracy of the network or model. How much to adjust the weights and biases according to the loss, is solely determined by calculating gradients of the loss function with respect to the weights and biases by chain rule. The gradient is calculated backwards through the network giving rise to the name backpropagation. The steps for calculating will be done in a three step process. First we define:

$$\delta^L = \nabla_{z^L} \mathcal{L}(\hat{y}, y). \tag{9}$$

Because backpropagation works backwards, we have for the last output layer N:

$$\delta^N = \nabla_{z^N} \mathcal{L}(\hat{y}, y), \tag{10}$$

where the gradient of the loss function can be calculated using chain rule:

$$\nabla_{z^N} \mathcal{L}(\hat{y}, y) = \nabla_{\hat{y}} \mathcal{L}(\hat{y}, y) \circ (g^N)'(z^N), \tag{11}$$

where $(\sigma^N)'(z^N)$ is elementwise derivative with respect to z^N. Then from this we can use the chain rule to work backwards through all layers. For layers $\ell = N-1, N-2, N-3, ..., 1$ we let:

$$\delta^L = (W^{(\ell+1)^T} \delta^{(\ell+1)}) \circ \sigma'(z^\ell), \tag{12}$$

where $\circ$ represents element-wise product and σ denotes the activation function for the layer. Gradients for layer ℓ are then calculated as:

$$\begin{aligned} \nabla_{W^\ell} \mathcal{L}(\hat{y}, y) &= \delta^\ell a^{(\ell-1)^T} \\ \nabla_{b^\ell} \mathcal{L}(\hat{y}, y) &= \delta^\ell. \end{aligned} \tag{13}$$

Each parameter, weights and biases, of the network is then changed by:

$$\theta_{i+1} = \theta_i - \eta \nabla \mathcal{L}(\theta). \tag{14}$$

Note that equation 14 is the same as equation 5 since it is simply the gradient descent part of backward propagation. Equations 9-14 for calculating gradients are taken from [6].

G. Federated Neural Network

Federated learning is a distributed approach to machine learning that allows the training of a central algorithm using data from multiple local servers or nodes. In this paradigm, instead of sending raw data to a central server, which could raise privacy concerns and entail significant data transfer costs, the central server merely maintains a global neural model. This model is then shared with the local nodes, typically present on user devices or edge servers, which perform training on their respective local data sets in parallel. By keeping the training data decentralized, federated learning preserves data privacy and reduces the risk associated with sharing sensitive information.

After the local nodes complete their training rounds, they transmit only the updated model parameters, rather than the raw data, back to the central server. The central server then integrates these parameter updates, reflecting the collective knowledge gained from all the local nodes. This iterative process of training, communication, and aggregation continues, allowing the central model to improve over time without directly accessing or storing the local training data. Figure 3 shows a basic visualization of a central model communicating with four different local nodes.

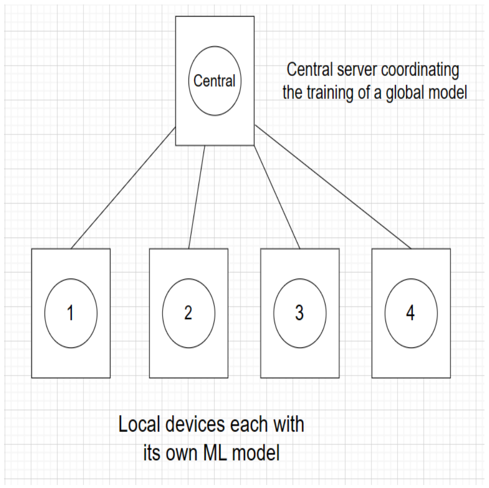
Central
Central server coordinating the training of a global model
1
2
3
4
Local devices each with its own ML model

Fig. 3. Illustration of central and local servers each with a neural network model.

Federated learning offers several advantages in scenarios where data privacy is a concern or when the data is distributed across various locations. Furthermore, federated learning reduces network bandwidth requirements by minimizing the amount of data transferred during the training process. This will be further improved by introducing stochastic quantization as is discussed next.

H. Stochastic Quantization

Following [2], the model is quantized before broadcast and update. The quantization function as described in the paper is the following:

$$Q(x_i, q) =$$
$$sign(x_i) * (x_{min} + (x_{max} - x_{min}) * \phi(\frac{|x_i| - x_{min}}{x_{max}x_{min}}, q)), \tag{15}$$

where q is a given quantization level ≥ 1. Then define

$$Q(\vec{x}, q) = [Q(x_1, q), Q(x_2, q), ..., Q(x_d, q)]^T, \tag{16}$$

for given $x \in \mathbb{R}^d$. ϕ is a quantization function defined as follows:

$$\phi(x, q) = \begin{cases} l/q, & \text{with probability } 1 - (xq - l) \\ (l+1)/q, & \text{with probability } xq - l, \end{cases} \tag{17}$$

where l is an integer $\in \{0, 1, ..., q-1\}$ such that x $\in [l/q, (l+1)/q)$ for $0 \leq x \leq 1$. The purpose of this is in order for transmissions between nodes to be faster and require less bandwidth while, hopefully, not severely affecting performance.

III. Methodology

A. Training and Test data

Training and test data is taken directly from MNIST. 70000 handwritten images of digits 0 to 9. 10000 of which is used for testing. The images are 28×28 pixels large.

B. Implementation in Python

All implementation was done in Python, only making use of the Numpy library. The Numpy library is called with np. The implementation of the deep neural network is done with object orientation in mind, where an overarching architecture of classes is the fundamental part of the implementation. This was done in order to make the implementation clear and easy. All the code used can be found in the appendix, but the following are the most important functions implemented.

1) Implementation of the Deep Neural Network: For the neural network we created a few classes, $FCNN$ which inherits $Network$ and $Fully\_Connected\_Layer$ which inherits $Layer$. In short, an $FCNN$ is a collection of $Fully\_Connected\_Layer$ objects, with added methods on how construct, train and test the network. This is done by implementing forward propagation, backward propagation and batch gradient descent, as explained in section II. The $Fully\_Connected\_Layer$ object is, in short, a weight matrix and bias vector with methods to assist forward propagation and backward propagation.

The $construct\_network()$ function below in Function 1 exists in $FCNN$ and creates a network of arbitrary size by creating a set amount of $Fully\_Connected\_Layer$ objects and initializing the parameters from a normal distribution.

Function 1. Construct network

```
def construct_network(self):
   layers = []
   for layer_type in self.layer_specs:
      temp_weight = np.random.randn
      (layer_type[1], layer_type[0])/2

      temp_bias = np.random.randn
      (layer_type[1], 1)/2

      layers.append(Fully_Connected_Layer
      (temp_weight, temp_bias, layer_type[2]))

   return layers
```

Function 2 in the $FCNN$ class implements forward propagation by utilizing function 3 in $Fully\_Connected\_Layer$.

Function 2. Forward propagation

```
def forwardprop(self, image):
   z = image
   a = sigmoid(z)
   for layer in self.layers:
      a, _ = layer.forward(a)
   return a
```

Function 3 calculates the activation of each layer according to equation 8.

Function 3. Activation of layer

```
def forward(self, input_vector):
   z = np.dot(self.weight, input_vector)
   + self.bias
   a = activation(z, self.activation_func)
   return a, z
```

Next, $train\_network()$ in Function 4, which is part of the $FCNN$, iterates through the set of shuffled training data and performs batch gradient descent, calling on Function 5 and 7, utilizing back propagation for a predetermined batch size and amount of epochs.

Function 4. Train network

```
def train_network(self):
   combined_training = list(zip(
   self.train_X, self.train_y))
   for epoch in range(self.epochs):
      random.shuffle(combined_training)
      for start_point in range(0,
      len(combined_training),
      self.batch_size):
         batch = combined_training[start_point:
             ↪ start_point + self.batch_size]
         self.__backprop_batch(batch)
         self.__gradient_descent(len(batch))
```

Function 5 contains a function $\_\_backprop\_batch()$ which uses Function 6 to perform back propagation on a batch.

Function 5. Backpropagation of batch

```
def __backprop_batch(self, batch):
   self.__reset_layer_gradients()
   for (image, y) in batch:
      self.__backprop(image, y)
```

The $\_\_backprop()$ function in Function 6 below performs back propagation throughout the layers in the neural network. Since the implementation is object oriented, the function exists within a class where layers objects. The function first saves the necessary values for the last layer in order to begin back propagation. It then iterates through every layer backwards, computing the delta value as shown in equation 12. It does this by calling upon the backward function depicted in Function 8. Note that the first layer of back propagation has to be handled separately, as described in Equation 11.

Function 6. Backpropagation

```
def __backprop(self, image, y):
   a_layers, z_layers =
   self.forwardprop_list(image)
   delta = self.__first_delta(z_layers[-1], a_layers
       ↪ [-1], y)
   self.layers[-1].first_layer_backward(delta,
       ↪ a_layers[-2])
   for i in range(2, len(self.layers) + 1):
      delta = self.layers[-i].backward(self.layers[-
          ↪ i + 1].get_weight(), delta, z_layers[-i
          ↪ ], a_layers[-i - 1])
```

Gradient descent function in Function 7 iterates through the layers of the network and utilizes a function called $descend()$ showcased in Function 10 in order to perform SGD on a given batch.

Function 7. Gradient descent

```
def __gradient_descent(self, batch_size):
   for layer in self.layers:
      layer.descend(batch_size,
      self.learning_rate)
```

Backward function depicted in Function 8 calculates the necessary back propagation values using chain rule according to equations 9-12.

Function 8. Backpropagation calculation

```
def backward(self, prevWeight, delta, prev_z, prev_a
    ↪ ):
   delta = np.dot(prevWeight.transpose(), delta) *
       ↪ activation_prim(prev_z, self.
       ↪ activation_func)
   self.grad_W += np.dot(delta, prev_a.transpose())
   self.grad_b += delta
   return delta
```

Function 9 calculates the backward propagation of the last layer in order to propagate through the rest of the layers.

Function 9. Backpropagation first layer

```
def first_layer_backward(self, delta, activation):
   self.grad_W += np.dot(delta, activation.transpose
       ↪ ())
   self.grad_b += delta
```

Function 10 showcases the function $descend()$ which changes the weight and bias parameters according to Equation 14 as explained in section 2. The function also takes into account the size of the batch by dividing the learning rate by the batch size.

Function 10. Changing parameters

```
def descend(self, batch_size, learning_rate):
   self.weight -= learning_rate / batch_size * self.
       ↪ grad_W
   self.bias -= learning_rate / batch_size * self.
       ↪ grad_b
```

2) Implementation of Federated Learning: We then extend the neural network by making it federated. This is done by creating a $Federated\_Learning$ object which has a $FCNN$ object and several $FCNN\_device$ objects. Federated_Learning changes several methods from $FCNN$, most notably the one concerning the training of the network. Then $FCNN\_device$ inherits $FCNN$ and implements methods to receive and transmit data. The core structure of a deep neural network remains consistent while central and local nodes are added. Denote the main model as $\theta(t)$ and the estimates at the local nodes as $\hat{\theta}(t)$ where t is a global iteration count. The algorithm, implemented directly from the paper, goes as follows:

Algorithm 1 Federated Learning

```
for $t = 0, 1, ..., T-1$ do
    Global Model Broadcasting
    PS broadcasts $Q(\theta(t) - \hat{\theta}(t-1), q1)$
    $\hat{\theta}(t) = \hat{\theta}(t-1) + Q(\theta(t) - \hat{\theta}(t-1), q1)$
    Local Update Aggregation
    for $m = 1, 2, ..., M$ in parallell do
        Device M transmits $Q(\Delta\theta_m(t) + \delta_m(t), q2) = Q(\theta_m^{\tau+1}(t) - \hat{\theta}(t) + \delta_m(t), q2)$
    end for
    $\theta(t+1) = \hat{\theta}(t) + \sum_{m=1}^{M} \frac{B_m}{B} Q(\Delta\theta_m(t) + \delta_m(t), q2)$
end for
```

The algorithm works through the following: given a quantization level q1 the central server/node transmits a quantization of the current model at global time step t. The devices acquire the model estimate:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + Q(\theta(t) - \hat{\theta}(t-1), q1), \quad (18)$$

where $\hat{\theta}(0) = \theta(0)$. The local nodes, each having received a quantized model, then perform τ-step gradient descent on the local training data through corresponding i-step:

$$\theta_m^{i+1}(t) = \theta_m^i(t) - \eta_m^i(t)\nabla F_m(\theta_m^i(t), \xi_m^i(t)), \quad (19)$$

where $\theta_m^1(t) = \hat{\theta}(t)$ and $\xi_m^i(t)$ denotes the local mini-batch chosen from local model training set B_m. Every local node then transmits back an updated local model, again quantized according to

$$Q(\Delta\theta_m(t) + \delta_m(t), q2), \quad (20)$$

δ_m is a quantization error updated as follows:

$$\delta_m(t+1) = \Delta\theta_m(t) + \delta_m(t) - Q(\Delta\theta_m(t) + \delta_m(t), q2), \quad (21)$$

$\delta_m(0)$ is set to zero. The central server, having received a quantized model from each local node, updates the global model as:

$$\theta(t+1) = \hat{\theta}(t) + \sum_{m=1}^{M} \frac{B_m}{B} Q(\Delta\theta_m(t) + \delta_m(t), q2). \quad (22)$$

The important functions for the code implementation of this is the global model broadcasting, local update aggregation and a function for updating the global model based on the local models. Function 11 showcases the function for broadcasting the main model to the local models where models are the local servers and main model is the central server.

Function 11. Global model broadcast

```
def global_model_broadcasting(self):
    if self.quantized_main_model is None:
        broadcast = self.main_model.
            ↪ quantized_network(self.q1)
        self.quantized_main_model = broadcast

    else:
        broadcast = (self.main_model - self.
            ↪ quantized_main_model).
            ↪ quantized_network(self.q1)
        self.quantized_main_model = broadcast +
            ↪ self.quantized_main_model

    for model in self.models:
        model.recieve_global_model(broadcast)
```

Function 12 showcases the algorithm for the local update aggregation where each local server or device transmits back the trained parameters.

Function 12. Local update aggregation

```
    def local_update_aggregation(self):
    local_updates = []
    for model in self.models:
        model.thread_process()
    for model in self.models:
        local_updates.append(model.transmit_update
            ↪ ())
    return local_updates
```

Function 13 shows the function for updating the main model based on the parameters received from the local devices.

Function 13. Central model update

```
def update_main_model(self, local_model_updates):

    local_update_sum = local_model_updates[0] *
        ↪ len(self.data[0])/len(self.models)
    for model_update in local_model_updates[1:]:

        local_update_sum += model_update * len(self
            ↪ .data[0])/len(self.models)
    local_update_sum = local_update_sum / len(self
        ↪ .data[0])

    self.main_model = self.quantized_main_model +
        ↪ local_update_sum
```

IV. Results

A. Training and Testing

As discussed in methodology, training and testing is done with MNIST data set. The important marker for how good the networks are, is the accuracy of which they are ale to predict what digit the image displays.

B. Fully Connected Neural Network

For the FCNN, layers are of the size 784, 64, 32, 16 and 10. We use sigmoid activation for all layers expect the last, which uses softmax. The loss function is set to cross-entropy. The architecture of the network is showcased in Table 1. Note that our implementation has the possibility of any number of layers, these sizes were chosen to compare to the results in [2].

TABLE I
SHOWING THE ARCHITECTURE OF THE FULLY CONNECTED NEURAL NETWORK

Layer	Size	Activation function
1	784	Sigmoid
2	64	Sigmoid
3	32	Sigmoid
4	16	Sigmoid
5	10	Softmax

The batch size is set to 500. In Figure 4 we see accuracy of the network compared to how many epochs it has trained and in Figure 5 the total loss is plotted against epochs.

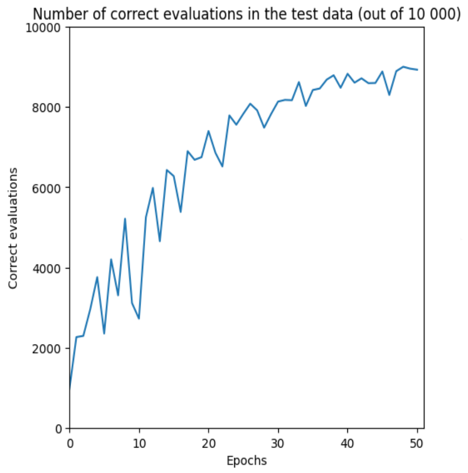

Fig. 4. Number of correct evaluations in test data for FCNN plotted against epochs trained.

Figure 4 shows that accuracy increases rather quickly and achieves a maximum of around 90%. Naturally, as accuracy increases the total loss decreases as seen in Figure 5.

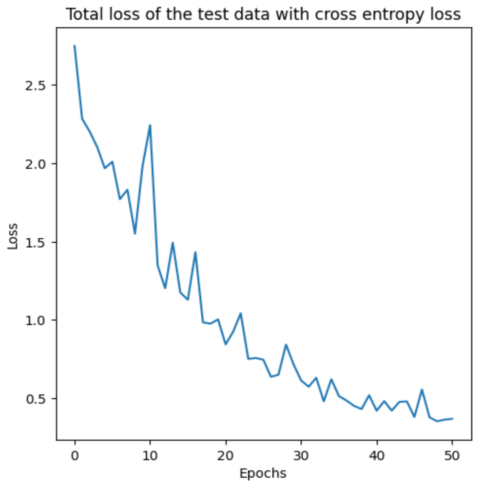

Fig. 5. Total loss value for FCNN plotted against epochs trained.

C. Federated Learning

Introducing Federated Learning, we aim to set our hyper-parameters as similar to [2] as possible. This is in order to compare the difference between our implementation and theirs, as well as being able to compare how federated learning affects a simple neural network. Thereby we set the amount of devices, M, to 40, the quantization levels q_1, q_2 both set to 2, τ set to 4 and learning rate set to 1. Note that the network in [2] still differs significantly, mostly in their use of several convolutional layers. The rest of the hyperparameters are identical to the FCNN above. Figure 6 shows number of correct evaluations in the testing set plotted against epochs.
Figure 7 shows the total loss value plotted against epochs.

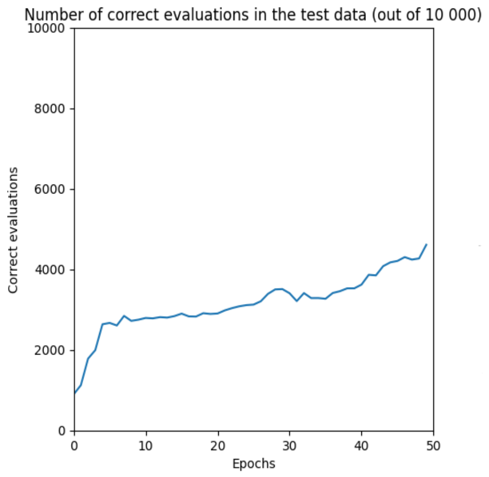

Fig. 6. Number of correct evaluations in test data for federated network with quantization plotted against epochs trained.

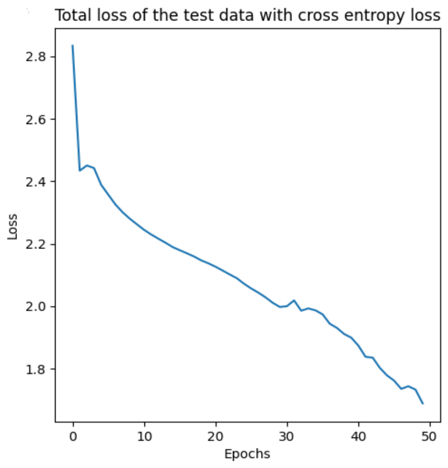

Fig. 7. Total loss value for federated network with quantization plotted against epochs trained.

Compared to the FCNN, the accuracy does not nearly achieve the same accuracy. After 50 epochs we achieve an accuracy of around 45%, although its rate of increase in the first epochs is similar to the FCNN case. We also compared how the network performed without quantization. Which is shown in Figures 8 and 9. The network does not seem to perform better without quantization. In Figure 8 and 9 the plots are more smooth compared to Figure 6 and 7. Seeing as our accuracy is significantly worse for the federated case compared to FCNN,

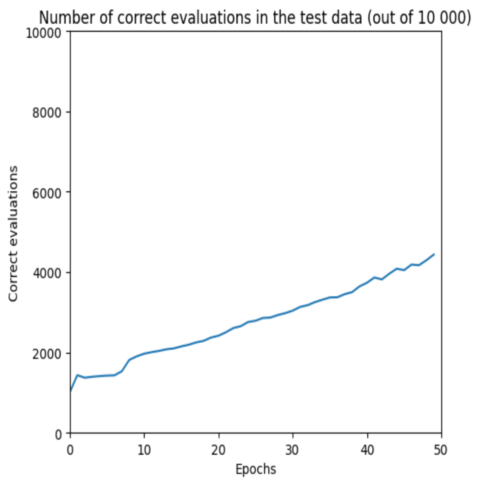

Fig. 8. Number of correct evaluations in test data for federated network without quantization plotted against epochs trained.

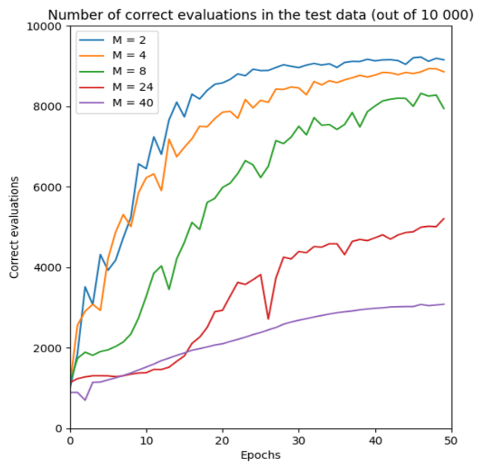

Fig. 10. Number of correct evaluations in test data for federated network without quantization for 2,4,8,24 and 40 devices plotted against epochs trained

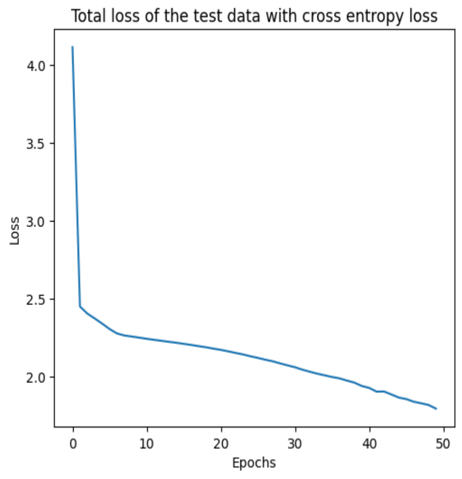

Fig. 9. Total loss value for federated network without quantization plotted against epochs trained.

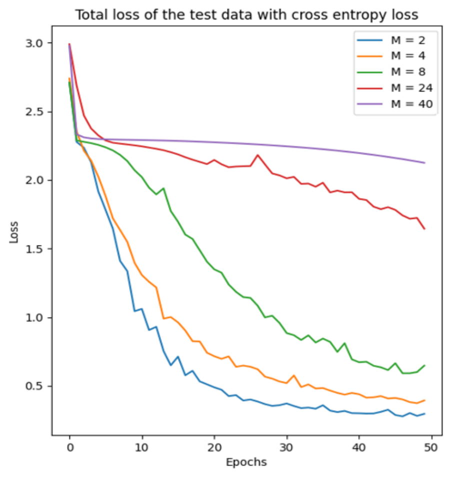

Fig. 11. Total loss value for federated network without quantization for 2,4,8,24 and 40 devices plotted against epochs trained.

we decided to investigate convergence dependant on number of devices without quantization. Figure 10 depicts the number of correct evaluations in the test data for 50 epochs for 5 different amounts of devices. Figure 11 shows the same but for the total loss value instead. For figure 10 and 11, we kept the rest of the hyperparameters identical as to before. Observe that when the amount of devices is small, the convergence and accuracy closely match the FCNN. However, as the amount of devices increases performance is significantly affected.

V. Discussion

Our fully connected neural network performs well with a high degree of accuracy. It is, however, clear that improvements can be made in the implementation. This is because it is purposefully simple. The most significant improvement we can make is probably implementing convolutional layers. Beyond that we can expect improvements if regularization, momentum, and training data pre-processing is implemented. Furthermore, because of computing limitations, the FCNN is comparatively small, better results are expected from larger networks. This can be seen in [7], where they achieve an error rate of only 0.35% with similar methods, though with a much larger network, a shrinking learning rate and extensive data pre-processing. For the federated case, we got worse results in accuracy and loss compared to [2] which had almost 90% accuracy after 50 epochs. Most notable difference between their implementation and ours is that they implemented convolution and used ReLU activation which explains some of the difference. They do not comment on whether they used regularization or momentum, though that is fairly standard practice. We suspect that the reason for our poor rate of convergence is due to overfitting of test data, as each individual device only has access to 1500 data points in the $M = 40$ case. Because of this, each device may over-fit its data leading to a poor composition in the main model. To emphasise this, we also found a correlation that the performance of federated learning decreases significantly as the amount of devices increases. As seen in Figure 10, a low amount of devices does not perform considerably worse compared to the FCNN case in Figure 4. When devices are increased we observe a significant decrease in performance. We also observed strange behaviour in the $M = 40$ case where the loss would sometimes get stuck around 2.29 as can be seen Figure 11. This further correlated with significantly decreased performance as we can see a greater performance in Figure 8 and 9, where the loss decreased beyond 2.29. Note that these were performed with identical hyperparameters.

Furthermore, reflecting the results of [2], we see that even a high rate of quantization does not lead to a significantly decreased rate of convergence. For our results it does seem to make the convergence less smooth, making the accuracy and loss oscillate up and down significantly more. This seems reasonable since quantization stochastically scrambles the parameters to a smaller set of numbers. This does mean that one can expect a higher maximal convergence with quantization, as the noise induced could lead to the loss function breaking out of a non-global local optima. However, we lacked the computing power to observe this.

VI. Conclusion

The implementation performed as expected when performing conventional supervised learning on the MNIST database, getting a decent degree of accuracy. However, while our implementation successfully performed federated learning on the MNIST data set we clearly demonstrated that more sophisticated networks are necessary for federated learning to be a reasonable alternative. The quantization function did not significantly affect performance. Fortunately, the implementation we created is flexible and easy to improve. Layers and hyperparameters can be added and changed seamlessly and further extensions can be added without affecting the current existing structure.

Appendix A
Python code

Acknowledgment

The authors would like to thank our supervisor Seyedsaeed Razavikia for helping and guiding us throughout our project, while giving us the room to make the project into what we wanted it to be.

References

[1] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated machine learning: Concept and applications,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, Jan 2019.

[2] M. M. Amiri, D. Gunduz, S. R. Kulkarni, and H. V. Poor, “Federated learning with quantized global model updates,” *arXiv preprint arXiv:2006.10672*, 2020.

[3] P. Cunningham, M. Cord, and S. J. Delany, *Supervised learning*. New York: Springer, 2008.

[4] Ng, Andrew, “Cs229 lecture notes,” *CS229 Machine Learning*, vol. 1, no. 1, pp. 1–3, Stanford, California. 2000.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, ch. 5, pp. 96-108, http://www.deeplearningbook.org.

[6] Ng, Andrew. (2019) Cs229: Additional notes on backpropagation. Stanford, California. . [Online]. Available: https://cs229.stanford.edu/notes-spring2019/backprop.pdf

[7] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, “Deep, big, simple neural nets for handwritten digit recognition,” *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.