Federated Machine Learning

Karan Samani
1161081087
Final Year Project BSc. Computer Science (Hons.)
Supervisor: Dr. Derek Bridge
Department of Computer Science
University College Cork
March 31, 2020

Abstract

In this report we will see the motivation behind the need for a federated process for machine learning, a process that preserves privacy. A brief overview of how machine learning and neural networks work will be provided, which is required to understand how the federated learning process works. After doing so, we will be moving onto the implementing the federated approach proposed by Google.

Google's approach will be implemented from scratch along with an implementation using Tensorflow's Federated framework. The latter being used as a sanity check to confirm that the implementation from scratch is indeed correct, doing so by running a few experiments and comparing results. After doing so, we will move on to exploring extended approaches to the federated learning idea based on a weighted and selective approach on a global level and a user level.

Once the implementation has been explored, we will run several experiments to compare the approaches. These include traditional machine learning, federated learning and the several extended approaches that were explored.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed:.....

Date:.....

Acknowledgements

Thanks to Dr. Derek Bridge for being my supervisor and dealing with me over the last few months, pushing me to my limits and supporting me throughout the year. Ever since first year, I wanted to do my final year project with him and the experience of doing so has lived up to my expectations.

Thanks to the Alexander Baran-Harper's channel on YouTube where I learnt how to use use LaTeX.

Contents

List of Figures

1 Introduction

1.1 Background

Modern day edge devices have a wealth of data on them and have more than enough computation power to run complex calculations on them with ease. These devices can range from a personal computer to a smart phone. In a world where data is power, access to the data on these devices is highly advantageous.

Artificial Intelligence (AI) can be described as the ability for a system to show "intelligence". Intelligence, as Dr. Derek Bridge put it, is the ability for a system to act autonomously and rationally when faced with disorder, uncertainty, imprecision and intractability. Machine Learning is a branch of AI which is based on the idea of creating a model that recognises patterns from data to be able to solve problems. Neural Networks (Section 2.2) are an example of machine learning with Deep Learning being a subset of neural networks where the models used have more layers in them. This division can be visualised in Figure 1. To be able to do so effectively, one must have access to a lot of data. More data equates to a higher probability of having a more robust and better overall model. If a model can look and learn from more data, the chances are that it can generalise well, and that is the ideal goal.

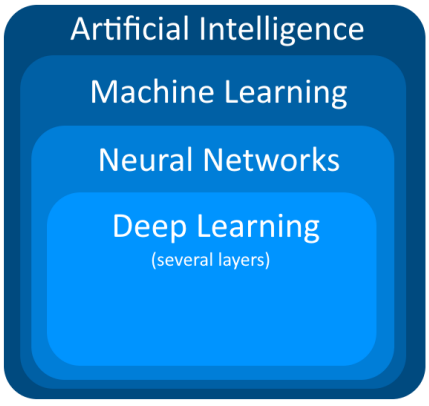

Figure 1: High level visual representation of what fields there are in the study of AI.

The solution to having well trained models seems straight forward. We should use the vast amounts data from the edge devices to train a model that, in theory, should be fairly accurate. To do so, the users must give their data to a server so that the server can then train the model on the data that was just provided to it by the edge users. This trained model can then be used by everyone to predict unseen samples. But there are some more complications. Users may not want to share their data but still want the benefits of having a model trained on everyone's data.

1.2 Motivation

Artificial Intelligence, specifically Machine Learning, is an idea that is taking the world by storm. A piece of software that would make machines autonomous. It was supposed to be better in every way, never getting tired, reaching at least the same competency levels as humans and quite possibly even better. It was hyped up to the point where the media started talking about AI putting people out of a job and eventually taking over the world. They even made far fetched references to the popular movie series Terminator.

Needless to say, this is not accurate. The idea of a general purpose AI, formally called Artificial General Intelligence, is no where near attainable. Current AI applications are good at specific tasks, and only those tasks because they have a narrow scope. Even with their narrow scope there are more topics that need to be addressed, such as the security issues that arise with better AI systems. One can use AI to create cyberweapons that can be used for hacking and spreading misinformation. Along with cyberwarfare, AI can be used to make traditional weapons more lethal. For instance, drones are now being used to target specific areas (and people) using ideas like image classification. At first glance this seems like a good idea as it should result in less casualties. But, AI systems are not 100% accurate so they could lead to an accidental strike. And if someone is able to hack into the system, they can make the drones target literally anything and anyone. Even in the right hands this technology can lead to disasters, let alone the wrong hands. Then there is the privacy aspect as well where people may not want to share their potentially sensitive private data under the fear that it can be misused. The idea of privacy will be the focus of this project.

1.3 Privacy Concerns

There are numerous examples where people may not want to share their personal data. For instance, for the training of a model that deals with predictive text, the input data would require essentially everything that a user may type into their device. It is pretty obvious to see why some people may not want to share the messages and other content that they type on their devices. It is a clear invasion of their privacy.

Another application could be training on images for classification purposes. People may not want to share images, which may include sensitive images, that they have stored on their devices with a third party. This can be extended to an even more sensitive topic of medical imaging where people may not want to share something like the X-Rays of their bodies.

In general, people are sceptical of sharing personal data. But there is still the need to train a model that has had exposure to as much data as possible.

1.4 Report Outline

The need for privacy was the motivation behind the idea of Federated Learning which was an idea proposed by Google in 2016 [4]. The idea of not having to share your data with someone else and yet still have the benefits of having a model that has exposure to their data will be the main point of interest in this report.

To start off, a brief overview of machine learning and neural networks will be provided in Sections 2.1 and 2.2 respectively. Then in Section 2.6 we will see how Google describe federated learning before discussing the design and implementation in Section 3.2. Following Google's approach, several extended ideas based on weighted and selective approaches in a central and peer-to-peer environment will be discussed and their implementation explained in Section 4. Along with that, outputs from the experiments to show how the extended ideas compare with Google's approach of federated learning and the traditional approach of machine learning in this context will be explained in Section 5.

2 Literature Review

2.1 Machine Learning

Machine learning is a very broad field which includes a lot of different learning methodologies. Learning can take place in a supervised context where the dataset is labelled, or in an unsupervised context where the data is not labelled.

In unsupervised learning, the only input data are the features and there are no input-output mappings. The aim is to find structure within the dataset and can also be useful in finding anomalies in the dataset. The most well known algorithm for this purpose is k-means++ clustering.

In supervised learning, the task is to learn a function that maps an input to an output, based on sample input-output pairs. In this project, the focus is on supervised learning where the aim is not to find structure within a dataset but rather trying to learn how to map the input data to a desired output. There are quite a few methods of finding the right function that fits the dataset, ranging from simple functions to very complicated functions.

One of the simplest approaches for supervised learning is called Linear Regression, which aims to solve regression problems. The data that is provided to it are pairs consisting of input features (as a vector) and the desired output. Based on the set of input pairs provided, linear regression tries to find a linear function, which is a set of coefficients β for all the features, that fits the dataset well. The set of input pairs provided is called the training set. To find the best possible function to fit the data, the idea of a loss function is used. This essentially says how distant the predictions are from the desired output. Loss in this case is usually mean squared error (MSE). The error e, is the difference between the actual value and the predicted value.

$$\text{MSE} = \frac{1}{n}\sum_{t=1}^{n} e_t^2$$

The values of β that fit the dataset the best would be the one with the lowest value for MSE on the training data. A naive way to solve this problem would be to iterate over all the infinitely many βs and run predictions on the n samples in the training set to give an output. We then use use the

predicted output and the desired output to calculate the loss values for all the β combinations. The β values that give the lowest MSE value (loss) would be chosen as the solution. But there are more sophisticated methods of solving this such as gradient descent. The latter can intuitively be thought as taking, usually small, steps in the direction that reduces the loss.

Logistic Regression follows the same basic principle as linear regression but is used for classification purposes instead. Classification problems are about predicting if a sample belongs to a certain class or not. The input data for this is similar to linear regression with it being a pair of input features (as a vector) but the desired output being a label representing a class of objects (like "dog"). Logistic regression still works off of building linear models using β under the hood and predicts numbers that are probabilities of a certain input being part of a certain class. For the prediction, input features are passed through a sigmoid function σ (also called the logit function) which outputs a number between 0 and 1, lets call this h_β.

$$\sigma(z) = \frac{1}{1 + e^{(-z)}}$$

$$h_\beta(x) = \sigma(x\beta)$$

These numbers are interpreted as the probability of the input being part of a class, usually the positive class (a class which requires action and is labelled as 1). Based on the probability, the input is classified to a class $\hat{y}$.

$$\hat{y} = \begin{cases} 0 & \text{if } Prob(\hat{y} = 1|x) < 0.5 \\ 1 & \text{if } Prob(\hat{y} = 1|x) \geq 0.5 \end{cases}$$

The idea of reducing the loss still applies, but a more complicated loss function is used in this case. Linear regression and logistic regression are both based off of a linear function so they cannot deal with complex data very well. Decision trees are an alternative that can better fit complex datasets and can be used for both regression and classification problems. They are more intelligible compared to other approaches. At a very high level, the structure of the tree dictates what path a sample input should take. A very simple decision tree can be seen in Figure 2.1. The inner nodes in the tree split the data based on conditions and the leafs represent the decisions made on the samples. A different loss function is used to optimise the answer.

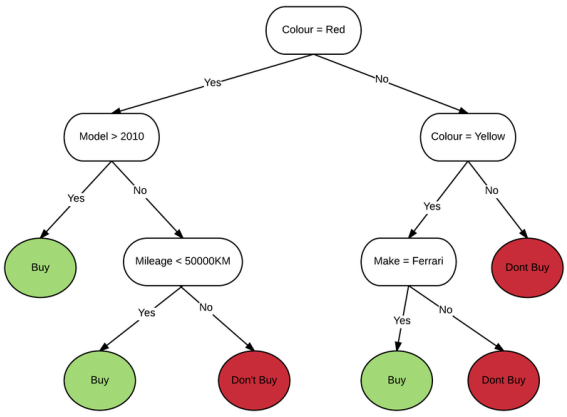

Figure 2: A very simple decision tree [5].

Neural networks have become the go to solution in recent times for pretty much all problems. They can cater to a wide range of problems, including very complex problems such as image classification, image localisation and natural language processing. They generally perform pretty well on those tasks too. There are also ways in which privacy concerns can be addressed when using neural networks. This is why they will be the only thing we use in this project.

2.2 Neural Networks

Neural networks, as seen Figure 1, are a subset of machine learning. Neural networks are not a new idea, they have been around for decades. But they only really took off in recent years with the emergence of better and more affordable hardware. The basic idea behind the workings of a neural network are quite straight forward. A model is defined and data is passed through it to make predictions. Then, based on the loss, some adjustments are made to the parameters learnt. This whole process is repeated by iterating over the dataset a set number of times during the training process. To start off, the idea of a neuron must be explained which are the basic building blocks of a neural network.

2.3 Neurons

A neuron computes a weighted sum of its inputs, passes it through a function (called the activation function) and outputs a value to be used later. The inputs received are from either the input layer neurons or the hidden layer neurons. The activation function used below is a step function that outputs a 1 if the weighted sum is more than a certain value (0 in this case), otherwise it outputs a 0. The weight w_0 for the input data point labelled 1 in Figure 3 is used to represent a bias. Because the input is 1, multiplying it with a weight w_0 is guaranteed to give a value which will act as a number that is always used in the summing process later before the value is passed into the activation function.

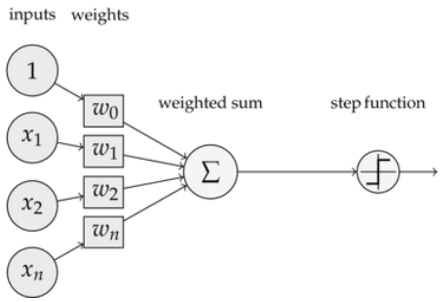

Figure 3: Major components of a neuron [3].

Note that, for brevity, the weighted sum can be written in a vectorised format. The weighted sum also includes w_0 which represents the bias.

$$w \cdot x \equiv \sum_j w_j x_j$$

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x \leq 0 \\ 1 & \text{if } w \cdot x > 0 \end{cases}$$

The activation function can be swapped out from the step function that was being used earlier with some other activation function such as ReLU (Rectified Logic Unit), sigmoid function, etc. ReLU takes the max of either 0 or the weighted sum and uses that as its output. This can be seen in equation 1.

$$\text{output} \quad = \quad \max\{0, w \cdot x\} \tag{1}$$

2.4 Architecture

In a neural network, there are layers of such neurons. These are usually broken down in three parts, the input layer, the hidden layer(s) and then the output layer. The input layer is not actually a layer of neurons but rather just a representation of the input data as a layer that connects to the hidden layers. The hidden layers can contain any number of layers with any number of neurons for the layers.

After the hidden layers is the output layer. This layer is responsible for outputting the predictions. The output layer usually has its own activation function which depends on the application, i.e., if it is a classification problem or a regression problem. Since this project focuses on multi-class classification problems, we will be using the softmax activation function for the output layer.

The layered structure described above is called the architecture of the model. Some of the common used architectures are densely connected layers (Section 2.4.1) and convolutional layers (Section 2.4.2). More information on the aforementioned and more architectures can be found in the book about Deep Learning by F. Chollet [2].

2.4.1 Dense

These are one of the most straight forward architectures and are generally used as the output layer of most neural networks. They are quite useful when placed as the last few layers as well, especially for classification purposes. In these, all the neurons are connected to every neuron in the subsequent layer. The input for these layers are flattened data, which can be thought of as a list of input data where nested lists are not allowed. An example can be seen in Figure 4.

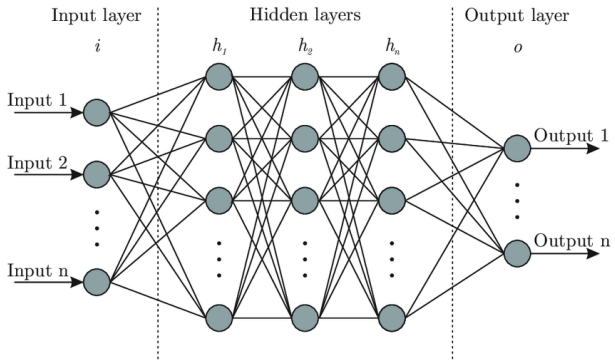

Figure 4: Basic dense neural network [1]

2.4.2 Convolutional

These are more complicated than the previously mentioned dense layers. The input data here is structured and not flattened. Their basic idea is to find patterns in the input data and make them more abstract as the layer count increases. They indicate the presence of certain shapes. The more common use for convolutional layers is in image classification.

A convolutional layer has a window (called the kernel) that is used to look over the input data and recognise patterns localised in that window. Every layer has a number of sub-layers which can be thought of as the number of patterns that the layer is trying to recognise and they are called the feature maps of the layer. For example if the depth is 3, the convolutional layer has 3 feature maps and will look at 3 kinds of patterns. The neurons in the following convolutional layer are connected to every neuron in the window of the previous layer, including the sub-layers as well. The set of weights that connect a neuron to the sub-layer neurons in the following layer are the same within the window. Figure 5 does a good job of giving a visual representation for the same.

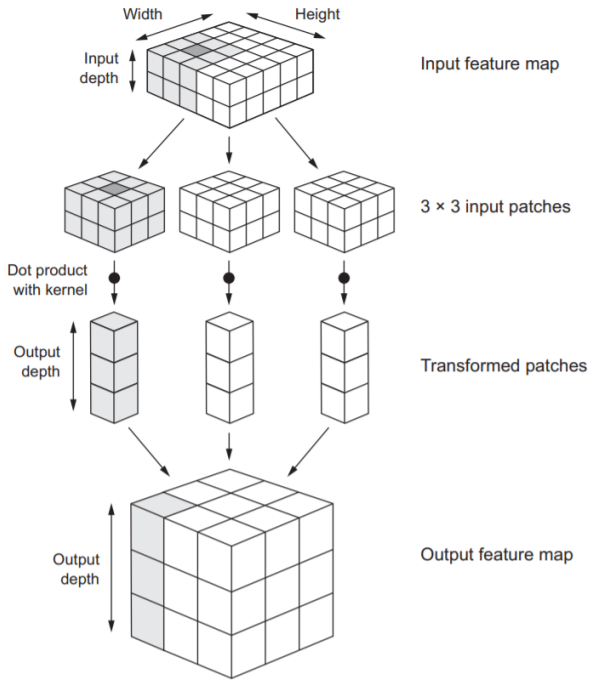

Figure 5: How a convolution works [2].

Convolutional layers are often followed by some maxpooling layers, which reduce the output size from the convolutional layers. This can be done for many reasons, such as saving memory and making the computations required less intensive. At the end of a series of convolutional layers, there is a series of dense layers that are used to give the predicted output(s).

2.5 Training

For a neural network to work, the weights with which the neurons are connected need to be tweaked and the process of doing so is called training the model. The model is trained on a training set, which is a subset of the whole dataset. Additionally, a validation set can be provided to see how the model performs on unseen data that is not the testing data. The progress is seen by comparing metrics for the model on both the training data and validation data over the course of the training. This is done to avoid leakage between the testing data and the training data. We will now go through the high level algorithm with which neural networks are trained. More information on this and the idea of neural networks can be found in the (online) book "Neural Networks and Deep Learning, Michael A. Nielsen" [6] and the book "Deep Learning with Python, Francois Chollet" [2].

The model is initialised with random weights before training begins. After that, the training data is then through the model to make predictions. These predictions are then compared with the actual values, and the loss is calculated using a loss function. For regression, mse is used and for classification sparse categorical crossentropy is used. The details in which they work are not required, but the idea of finding out how far off the predictions were from the actual value still holds. Using an methods like gradient descent or Adam (based on gradient descent and other methods, an algorithm called back propagation tweaks the weights in a way that would hopefully reduce the loss. These updates can be made based on every example, batches of examples or the entire dataset as a whole. Generally, the batched approach is taken and the entire dataset is broken down into batches of training data.

This whole process of predicting, calculating loss and tweaking the weights can be done several times by iterating over the whole training set several times. The number of times that the entire training set has been gone over is called the number of epochs that the network was trained on. This needs to be tweaked manually as it could lead to under-fitting or over-fitting the model. Which means, the model is too general or too specific, respectively, to be useful in actual usage.

2.6 Federated Machine Learning

Traditionally in ML, a server would have access to all the data. We call this server the central agent C. The data on the central agent is collected from a set of upto n edge users which we call U. Each edge user i of U, represented as U_i, sends their data to the central agent. User i's data is represented as D_i. Using all the data C has access to, $D = \bigcup_{i=1}^{n} D_i$, it would train a single model M which then anyone could then access to make predictions.

In Federated ML, D_i remains with U_i and instead only the weights of the user's model are shared. This means that at any give point, C does not have access to any data D_i from user U_i. By doing so, the privacy of the user's data is maintained. The training process in federated learning is performed on U_i's device itself. To be able do so, an identical copy of M is sent from C to every participating U_i. Every U_i then treats the model M that they received as M_i. This means that every M_i has the same architecture and the same randomly generated initial weights W_i. The weights are associated to the links that connect the neurons of the neural network as seen in Section 2.2.

Once U_i receives M_i, it can start training its M_i on its local data D_i. The training process for M_i generally starts based on certain conditions being met such as the device being plugged in for charging, WiFi connection, usage, etc. After the local training process for U_i has been completed, M_i would have new weights W_i. These weights would have been learnt and set such that they fit the user's D_i relatively well. User U_i then sends its learnt weights W_i to C for the averaging process to take place. The algorithm for the user side operations can be seen in Algorithm 1. Instead of sending all the weights, an alternative would be to send only the changes made to the weights, which is what Google does.

Algorithm 1: User side processing

```
1 def user(new_weights):
2     if new_weights != None:
3         M_i.set_weights(new_weights)
4     for e in local_epochs:
5         M_i.train()
6     M_i.send_weights_to_server() return M_i.weights
```

When C receives a set of upto n weights W_i from all the participating users, it uses them to calculate the average of the set of weights W_{avg}. The averaged weights W_{avg} are then sent from C to every U_i in U. This can be seen in Algorithm 2.

Algorithm 2: Server side processing

```
1 def central_agent(set_of_weights):
2     W_avg = average(set_of_weights)
3     send_to_all_users(W_avg)
```

The averaging process is the reason why we need to use the same architecture for all models M_i. Without the same architecture, the shape for every W_i would be different and therefore we would not be able to calculate an averaged W_{avg} that would be compatible with every M_i. The averaging mentioned in in Algorithm 2 line 2 is calculated by summing all received W_i and dividing by the number of W_i received, as seen in equation 2.

$$W_{avg} = \frac{\sum_{i=1}^{n} W_i}{n} \tag{2}$$

After receiving W_{avg} from C, every U_i replaces the weights W_i for their M_i with the new averaged weights W_{avg}. When the users have set the weights of their model M_i to be W_{avg}, they can start the training process again. This back and forth of training, averaging and training again can take place several times. Every time U_i trains their M_i and shares their W_i with C to receive W_{avg}, it is called a *round* of federated learning. After a few rounds, the resulting metrics can be pretty close to the metrics obtained with the traditional ML approach. Although, it must be noted that depending on the context the results may vary a lot and may require some changes as well. We will witness this in the experimentation section (Section 5) later in this report.

Algorithm 3 and Figure 6 provide a high level representation of the federated learning process. It starts off with C sending the initial model to the all the users. This model is the same M that was initialised with random weights in the traditional approach. It is provided to the algorithm as an input. After the users receive the model, for every round of federated learning, all the users set their weights to the new averaged weights (if provided), run local training and then share their current weights with C. C would

then calculate the new averaged weights and broadcast it to every U_i and the whole process takes place again for a set number of rounds. The calls made in lines 7 and 9 are calls to algorithms 1 and 2 respectively.

Algorithm 3: Federated Learning Core overall algorithm

```
1 def federated_learning_core(model):
2     central_agent.send_to_users(model)
3     new_weights = None
4     for round in rounds:
5         weights = [ ]
6         for user in users:
7             user_weights = central_agent(new_weights)
8             weights.add(user_weights)
9         new_weights = server(weights)
```

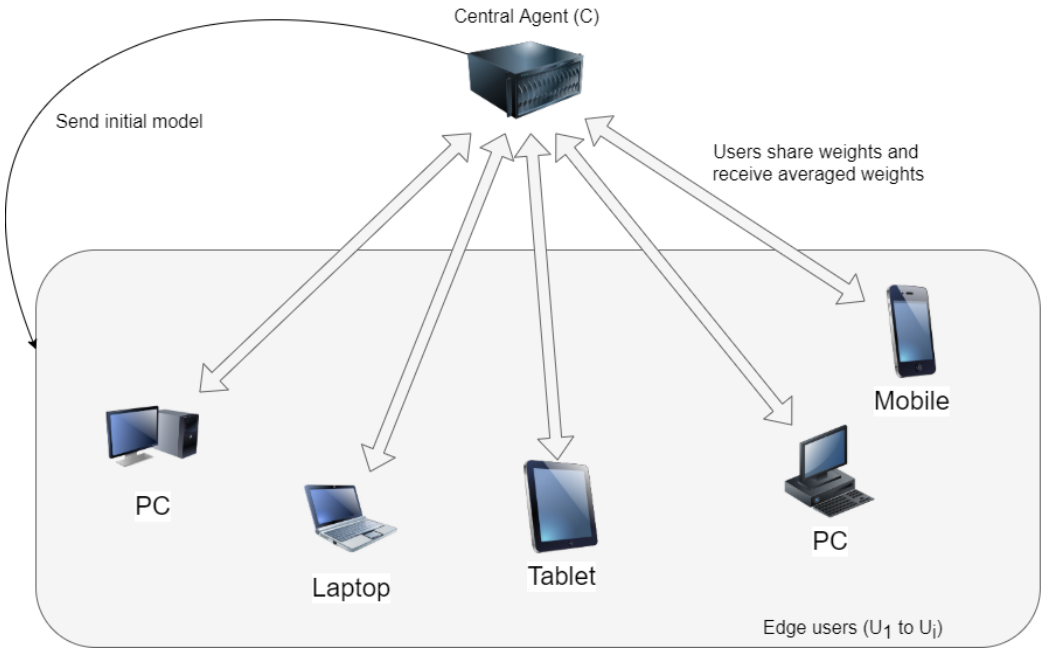

Figure 6: Federated learning.

2.6.1 Benefits

Federated machine learning has a few benefits. Firstly and most importantly, all the training takes places on the only the edge devices. This means that the users do not have to share their data with anyone. The only data they share are the parameters learnt from the training process that took place on their local data. And it is impossible to recreate the original data from just the parameters. Add to that the idea of Secure Aggregation [7] and one can very confidently say that the idea of privacy is held up to the highest standard. Secure aggregation is where the data is aggregated in stages. Instead of all averaging being done at the server where some data can possibly be reverse engineered, there are intermediary steps where the data is averaged and then the central agent finally averages those averages. Sometimes, noise is also added during this process. But we will not be focussing on secure aggregation in this project.

Secondly, the fact that all the training runs on edge devices means that there is no need for an investment in building a large training infrastructure by a company. The edge devices will do all the hard work of the model training and share the results which a central agent would then use quite trivially. So this is a better use of resources as a whole as idle devices would not just stay idle and would take part in the training process.

2.6.2 Drawbacks

As mentioned before, the federated approach can lead to similar performance as the the traditional approach. But this depends on the distribution of the data as we will see in later sections (5.7. If the dataset amongst all the users is very similar, then the overall result of the federated process can be very similar to the traditional approach. But if the users have skewed datasets, as is often the case in real life, the traditional approach is generally better. It is a trade-off of privacy vs. model performance that must be decided upon when deciding between the federated approach and the traditional approach respectively.

3 Core Design

In this section, the design and implementation aspect of the basic approaches will be discussed before moving on to discussing the extended ideas explored in this project.

3.1 Non-Federated Learning

Before talking about federated learning, we first need to set in place the traditional machine learning approach. For this, the idea of an imaginary user is utilised. We refer to this user as the *global_user*. The *global_user* assumes that we have access to data from every user. We set the metrics from this approach as *the* benchmark to beat. If any federated approach can be close to these metrics, then we can say that we can preserve privacy whilst maintaining a similar level of performance.

3.1.1 Design

This approach is quite straight forward in terms of what is to be done. The data is readily available to be used, so the first first step is to pass the training data into the model for training. The second one is to record how well the model performs on the training and validation data for every epoch of training.

With the dataset readily available for use, we need to select a model that fits the dataset well. We will look at how to find a dataset specific model in Section 5 where we perform experiments on different datasets. Given that the data and model are now ready to be used, we can pass the training data into the model to train it. The training process described in Section 2.5 is carried out and the model is trained on the training data for a number of epochs. To ensure that we can review the progress of the model, we also make sure that we evaluate the model between every epoch of training on the validation and testing data. This can then be plotted on a metric versus epoch graph to visualise the model's performance over time. For the final performance score of the model, we can evaluate the model after the training process has been completed on the completely unseen test data.

3.1.2 Implementation

With the approach being straight forward, its implementation is quite trivial. To be able to discuss the implementation, an overview of the technologies used in this project is required. The language of choice for this project was Python and the biggest reason for choosing it was the Keras library for deep learning. In this project, it uses with TensorFlow as its back-end to perform calculations. Keras provides a user friendly API to use Tensorflow to build neural networks with ease in Python. We chose to use TensorFlow as Keras' back-end because we will be using TensorFlow Federated (TFF) in Section 3.3 as a sanity check to confirm that our implementation of federated learning matches that of Google. TFF provides a framework which allows people to build a federated learning system without having to deal with the low level details. Another library used extensively was Matplotlib. Matplotlib is a Python library used for plotting graphs which. It allowed us to review the learning progress of the neural network and be able to compare the performance of different approaches explored in this project. Pandas and numpy were two more extensively used libraries that were used for data representation throughout the project.

The model M that is used here is initialised in Keras. The initialisation process is essentially selecting an architecture that works well on the dataset and then compiling it for further use, like training. Keras models require the input for the training process to be data in one of either two formats, pandas DataFrames or numpy arrays. We choose the latter as the way to store the data that we have to make it easier to pass data into the model for training and evaluation. Keras also requires the features of the data to be separate from the label associated with a sample. The low level control on the numpy arrays also allows us to perform more specific tasks, such as this separation of features and labels, on the dataset as and when required.

To train the model, we use the API that Keras provides for its models. A method called `fit` is associated to the model object which is used to train the model. The arguments passed into the method are the training data, validation data, epochs and the batch size. The training data is used to train the model. The validation data is used to gain some insight into how the model performs on data that is not the training data. Epochs is the number which specifies the number of times the training process iterates over the

entire training data. The batch size dictates the number of samples in the training data that must be processed before changes to the weights of M can be made. A simple call to this method on the model object will result in our model being trained on the training data.

We also need to record the performance of the model over time to be able to plot graphs using matplotlib. Keras, whilst training, maintains a history of the metrics obtained between epochs on the model object. Between epochs, the model is evaluated on the training and validation data, and the result is stored in a dictionary in the model object. This dictionary can be accessed to get what is essentially the progress report for the model on the validation and training data. We use this data to plot the graphs to see how well the model is performing.

3.2 Federated Learning

With the non-federated learning approach in place, we can discuss the federated approach in a similar way. This approach is exactly as explained before in Section **??** and as such, this section will not include a lot of details which were explained previously.

3.2.1 Design

In any federated approach explored, there is an obvious need to simulate users that take part in the process. But there is no point in implementing a network of users when the idea of users can be simulated on just one machine. These simulated users have one main goal, they must not be able to access data from other users. We needed to come up with a way to simulate users who only have access to their local data and their local model. In other words, a way where only U_i can access M_i and D_i. To do so, we chose an objected oriented approach where the users would be represented as objects U_i. These objects would contain all the information relevant to a given user and nothing more. This means that user object U_i would have access to M_i and D_i. It is to be noted that for a given dataset, all user models M_i are the same as model M which was selected for the $global_user$.

The core federated learning algorithm used here is the same as Google's algorithm 3 described in Section 2.6. But it's design was altered just a little bit to allow us to review the progress of the federated learning process and accommodate our for choice of not representing C as a distinct user.

The decision to not represent C as an object was taken because its functionality could be incorporated in the logic of the algorithm itself. This is done by replacing line 9 in algorithm 3 by line 2 in algorithm 2. As for being being able to review progress at the end, we had to store evaluation metrics of the model on the test data before and after the local training process. These evaluations were done during every round of federated training. We call these evaluations pre_fit and $post\_fit$ evaluations respectively. The pre_fit evaluations allow us to see the model's performance with the averaged weights and the $post\_fit$ evaluations allow us to see the model's performance with the weights after local training is performed. This helps us see how well the averaged weights and post training weights fit an arbitrary user U_i's test data D_i.

We augment algorithms 3 and 1 to reflect these changes which results in algorithms 5 and 4. The averaging in line 9 still uses equation 2.

Algorithm 4: User side processing

```
1  def user(new_weights):
2      if new_weights != None:
3          M_i.set_weights(new_weights)
4      evaluation = user.evaluate_model()
5      user.add_pre_fit_evaluation(evaluation)
6      for e in local_epochs:
7          M_i.train()
8      evaluation = user.evaluate_model()
9      user.add_post_fit_evaluation(evaluation)
10     return M_i.weights
```

Algorithm 5: Federated Learning

```
1 def federated_learning_core(model):
2     send_to_users(model)
3     new_weights = None
4     for round in rounds:
5         weights = [ ]
6         for user in users:
7             user_weights = user(new_weights)
8             weights.append(user_weights)
9         new_weights = average(weights)
```

The algorithm starts off by sending the model M to every user. Each user then refers to their model as M_i. Then, before the model is trained by the user, it is evaluated on the users test data and that information stored in the collection of pre_fit evaluations for the user. We can do the evaluation on test data without the fear of leakage because we do not use the results to make changes to the model. Every user U_i then trains their model on their training data to get the updated weights W_i for their M_i. After the training, the evaluation is conducted once again on the test data and then stored in the collection of $post\_fit$ evaluations. The updated weights W_i are returned by every user and stored in the main algorithm. This set of weights is passed in to the averaging function that returns an average of the weights called W_{avg}. The averaging function is based on equation 2. These weights are then passed onto the users in the next round of federated learning, where they initialise their models with the new averaged weights and conduct the whole process again. After a set number of rounds, the federated learning process ends. In this project, we decided to use a high number so we could observe how the process runs over a long period of time.

3.2.2 Implementation

The first thing that had to be implemented was the User class. It is used to store all the data local to a user. Instances of this user class are used to represent an arbitrary user U_i. The data stored on U_I includes the training, validation and test data, the model of the neural network and the history of the metrics obtained during the training of the model. The training, validation and test data are stored as numpy arrays. Numpy arrays were used over standard Python array lists because of their vectorised operations

which make them faster and more concise than using Python's array based lists. Another reason to user numpy arrays was because of the fact that Keras models require their input to be numpy arrays or pandas DataFrames. Storing the data as numpy arrays standardises the structure of the object for use throughout the project and increases re-usability of the code. The Keras model is stored in U_i as well along with the pre_fit and $post\_fit$ metrics which are stored in numpy arrays. The details of the completed class can be seen in Figure 7.

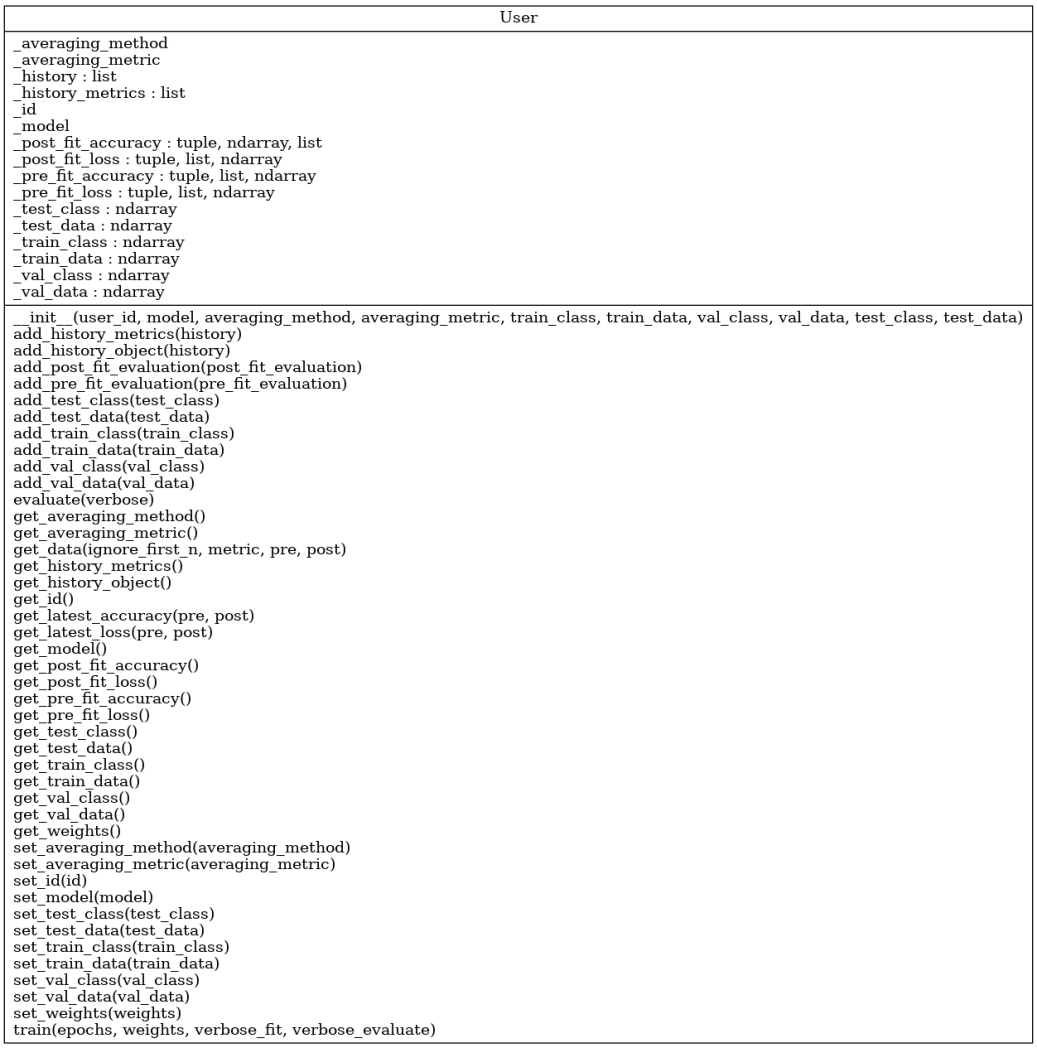

Figure 7: Methods and attributes of the User class.

Then for every user in the dataset, we must instantiate a user object U_i to represent them. The data relevant to the user is stored in this object after that. The details of this process will be explored in Section 5. After initialising the users, implementing the main algorithm was the next goal.

The algorithm starts off by sending M to all the users which would ordinarily be done in a networked fashion but since we are using simulated users, a simple for loop suffices. In the loop, a copy of the compiled Keras model M is stored in every U_i which the users then refer to as M_i. After doing so, the federated learning process can begin. This process was implemented as a function, `train_fed`, to increase re-usability throughout the project. This function would take in arguments such as `epochs`, `rounds` and more, to allow for flexibility in terms of how to run the training process. In it we conduct a set number of rounds of federated learning in which the training of the user's model and the sharing and averaging of the weights takes place.

To train all the user models, we iterate over all the users and call the `train` method on every U_i. In the `train` method, we make use of the API Keras provides for its models. Using the `set_weights` method of M_i, we can set the weights of the model to be the weights that we pass into the method. For the first round of federated learning, we do not set the weights to be anything new. But for subsequent rounds, the averaged weights W_{avg} are passed into the `set_weights` method which set the weight of M_i to be W_{avg}. After doing so, the model is evaluated on the test data by using the `evaluate` method of U_i. This method intern uses the `evaluate` method provided by Keras to run evaluation on their models on give data, test data in our case. This data is then stored in the numpy arrays dedicated to storing pre_fit metrics such as *accuracy* and *loss*. Then a simple call to the `fit` method of the model will train it on the training data that we pass into the method for a given number of epochs. After the training, we once again call the `evaluate` method of U_i to evaluate the model on the test data and record the metrics in the numpy arrays. As mentioned before, Keras also maintains history of the metrics during the training process in the model object. This history is reset every time the `fit` method is called. So, after every local training process we store the metrics from this history in U_i as well in a Python list. We can use this information to plot more graphs and review the progress of the learning process, but this time on a more granular level because this is a history of *epoch* evaluations. The pre_fit and $post\_fit$ evaluations we store are *round*

based evaluations.

Once all the users have been trained, their weights need to be averaged. To make the averaging process more modular, an class dedicated to averaging was implemented. This class is called Average. The details of the final version of the Average class can be seen in Figure 8.

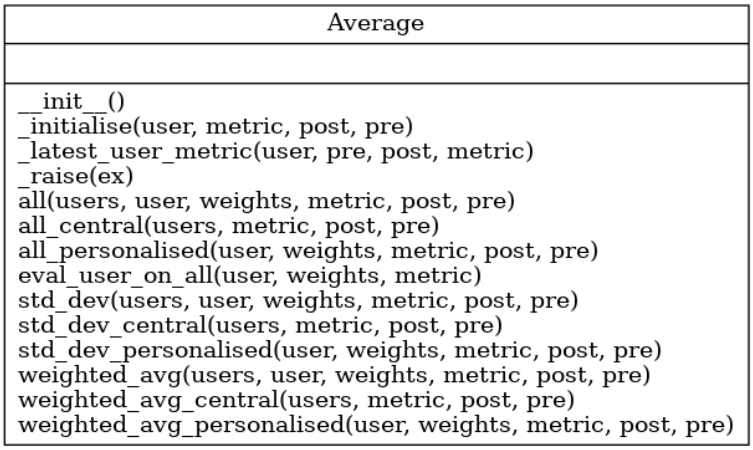

Figure 8: Methods and attributes of the Average class.

Residing in this class are static methods that deal with different kinds of averaging, one of which is the implementation of the averaging function in equation 2. The method is provided with a reference to all U_i in the form of a dictionary. With access to U_i comes access to model M_i for the user as well. The `get_weights` method associated to M_i is used to extract the weights W_i from model M_i. This is done for every user and then those weights are used to calculate the averaged weights W_{avg}. Keras models store their weights as a list of numpy arrays. To be able to use numpy's quick calculations we cast the list into a numpy array of numpy arrays which is still referred to as W_i. We also initialise a numpy array full of zeros with the same shape as W_i and name it *new_weights*. This is used to collect the sum of all the weights as we iterate over all the users to calculate the average. As we iterate over the users, we use the `get_weights` method to get the weights (which is a list of numpy arrays), cast it to a numpy array of numpy arrays and then add it to *new_weights*. The shape of *new_weights* never changes throughout the whole process. This means that at the end of the loop we can divide

new_weights by the number of users and get W_{avg}. W_{avg} is then returned into the main function `train_fed` where it will be fed back to all the users for the next round of federated learning where they initialise their models M_i with weights W_{avg}. After a set number of federated learning rounds have been performed, we can stop the process and plot graphs from the metrics we had stored along the way. Our choice of choosing a high number of rounds lets us know at exactly what point the metrics start plateauing, and we can then say that training after that point is not necessary.

3.3 TensorFlow Federated

In this section we will talk about simulating federated learning using TensorFlow Federated (TFF). The design aspect of this will not be discussed as it is the same as the federated approach we just discussed. Instead, we will focus on the implementation of federated learning using TFF. We implemented this approach for it to serve as a sanity check to see if our implementation as described in Section 3.2 is sound.

TensorFlow Federated is a framework for that allows for computation on decentralised data. TFF essentially has its own underlying language that can be accessed with Python. The building blocks of which are federated computations, which are computations that take place in a federated setting. TFF has two layers, the Federated Learning (FL) layer and the Federated Core (FC) layer. The Federated Learning layer allows high level plugging of Keras models into the TFF framework. We can perform basic tasks like the federated training process or evaluation without having to deal with the lower level concepts. The Federated Core layer provides us with low level control over the algorithms used in federated learning. In this section, we will be using the FL layer and not the FC layer as we do not need to implement custom algorithms.

TFF provides detailed documentation and tutorials on how to implemented federated learning. The implementation here is based on those tutorials. We will assume that the dataset here is preprocessed and ready to be used, details on how this was achieved for experiments can be seen in Section 5. The basic idea is that every user's data is to be put into a Dataset object that is part of TensorFlow's data representation library `tff.data`. The Dataset object corresponding to every user is then put into a Python list which will be used

going forward. A model architecture is then defined using Keras, but unlike the other approaches we have seen so far, we do not compile the model. This is done at a later stage in the process. A function is defined that constructs and returns a Keras model. We call it `create_model`. For models to be used in TFF, they need to be wrapped in a Model interface that is part of TFF's learning library `tff.learning`. This interface exposes methods that TFF needs to carry out training and other federated operations on the model in a federated setting. TFF will wrap the model for us by invoking a call to the function `tff.learning.from_keras_model`. We pass the non-compiled model from `create_model` into this function, along with other parameters like the loss function, metrics functions and a sample of the dataset. The sample of the dataset is provided for initialisation purposes. This process of constructing the model and wrapping the model is placed into a function as well. We call this function `model_fn`.

We can now initiate the federated learning process with TFF. A simple call to the function `tff.learning.build_federated_averaging_process` with `model_fn` as the parameter constructs a Federated Averaging algorithm and packages everything into a `tff.utils.IterativeProcess` which we call `iterative_process`. This constructs everything that needs to be done on the central agent and on the user's end, including compiling the model distributing them to users. It essentially implements everything that takes place in a round loop iteration in algorithm 3. We can now initialise `iterative_process` by calling the `initialize` method on it which returns the server state which we call `state`. A single round of federated averaging is performed by making a call to the `next` method on the initialised `iterative_process` object. We pass `state` and the training data into the `next` method which returns the updated server state `state` and the metrics at the end of the round. The method call causes all the local training on the simulated users, sharing and averaging of the weights and reinitialising the users models with the averaged weights to take place. Running the `next` method for a number of times representing the rounds of federated learning we want to perform.

At the end of the training, we must evaluate the model. For that we make a call to `tff.learning.build_federated_evaluation` and pass in the `model_fn` we defined. This initialises the federated computations that need to take place for this process returns the corresponding federated computation

object. We pass in the trained model and testing data into this to receive the metrics of the evaluation of the model on the training data. The trained model containing the averaged weights can be accessed from the server state that is returned after a round of federated learning. These metrics represent the performance of this implementation of federated learning using TFF. The comparison of the TFF implementation and our implementation of federated learning can be seen in Section 5. The results showed that the two implementation did not differ massively, which meant that our implementation was logically sound and that we could use it for further experiments.

We decided not to use the TFF implementation because of the learning curve required in working with the FC layer of TFF. It was more intuitive to work with a sound implementation that we had built up from scratch instead of having to hack together our extended ideas using the TFF framework.

4 Extensions to Federated Learning

In this section, we introduce several extensions to Google's federated learning idea that we conceived in the course of this project. These extensions are based on different averaging methodologies in a central and peer-to-peer averaging context. We will also look at an idea where the users do not share their weights with anyone.

4.1 Centralised Averaging

Google's version of federated learning is implemented in the context of central averaging. This is where a *central* agent C handles the process of averaging the set weights sent from a set of users. In Google's approach described in Section 3.2 the averaging is done in a neutral way where every participating user's weights are weighed equally and they are all included in the averaging process. This was previously illustrated in equation 2. We will now look at some ideas where that is not the case.

4.1.1 Weighted Average

Our first extension is weighted averaging. In weighted averaging, the idea is to weigh the weights of certain users more than others in the averaging process. The criteria on doing so are the latest evaluations that the users obtain when evaluating their model on their testing data. The evaluations provided by user U_i are referred to as E_i. Previously, the only thing that the users had to send to C were their weights W_i. But now, the users must also send their evaluation E_i to C, along with their weights W_i. This allows C to make decisions on which weights should be weighted more when computing the averaged weights.

For example, let us consider two users U_x and U_y with their latest evaluations being E_x and E_y. If the value of E_x is 100 and E_y is 50, where higher is better, then W_x will be weighted twice as much as W_y in the weighted average W_{avg}. To achieve that, we compute the sum of $W_x \times E_x$ and $W_y \times E_y$ and divide it by $E_x + E_y$. This results in a weighted average W_{avg} for users U_x and U_y where W_x is weighted twice as much as W_y in accordance with their respective evaluations. This can be generalised into an equation with

n users as seen in equation 3.

$$W_{avg} = \frac{\sum_{i=1}^{n} (W_i \times E_i)}{\sum_{i=1}^{n} E_i} \tag{3}$$

In the case where the choice of metric is such that a lower value is better than a higher one, such as in loss, we replace E_i by its inverse $\frac{1}{E_i}$ in equation 3. If in this case E_i is zero, then its inverse is not defined. There are several ways to deal with that, but in this project we chose to just use 10^{-6} as E_i and then take its inverse.

The design of the federated learning process with weighted averaging is very similar to what is described in Section 3.2 with only a couple of changes. The first change is that the user must also share their latest evaluations along with their weights. We change algorithms 4 and 5 to algorithms 6 and 7 respectively to reflect this change.

Algorithm 6: User side processing

```
1  def user(new_weights):
2      if new_weights != None:
3          M_i.set_weights(new_weights)
4      evaluation = user.evaluate_model()
5      user.add_pre_fit_evaluation(evaluation)
6      for e in local_epochs:
7          M_i.train()
8      evaluation = user.evaluate_model()
9      user.add_post_fit_evaluation(evaluation)
10     return M_i.weights, evaluation
```

Algorithm 7: Central Federated Learning

```
1  def federated_learning_core(model):
2      send_to_users(model)
3      new_weights = None
4      for round in rounds:
5          data = [ ]
6          for user in users:
7              user_weights, user_evaluation = user(new_weights)
8              data.append((user_weights, user_evaluation))
9          new_weights = average(data)
```

The second change is that we make line 9 in algorithm 7 use equation 3 instead of equation 2. Doing so successfully converts Google's federated learning process based on equal averaging into one that works on weighted averages instead.

As for the *implementation* of this idea, it was fairly trivial because of the way the framework was implemented in Section 3.2. The only thing to be changed in the implementation was the averaging function. It had to be swapped out with a weighted averaging function that is in accordance with equation 3. To do so, another static method is implemented in the class Average to compute the weighted average. This method is once again provided with a reference to all users in a dictionary data structure. With access to U_i we also get access to its model M_i and its latest evaluation E_i, both of which are stored in U_i. We start off by initialising a numpy array full of zeros with the same shape as W_i and name it $new\_weights$. We use this to collect the sum of the weights as we iterate over all the users to calculate the weighted average. As we iterate though the users, we get the weights of the user's model by using the `get_weights` method on M_i. We convert the returned weights W_i, a list of numpy arrays, into a numpy array of numpy arrays for quick calculations and concise code. We then retrieve E_i to be multiplied with W_i. In the case where the metric for E_i is accuracy, we perform a scalar multiplication of the weights with the accuracy. In the case where the metric for E_i is loss, we perform a scalar multiplication of the weight with the inverse of the loss using 10^{-6} as the loss if it is zero. We store the resulting weights by adding them to $new\_weights$ in every iteration. The shape of $new\_weights$ never changes throughout the entire process. We also keep a sum of all the evaluations E_i used, which we call E_sum. After iterating over all the users, we divide $new\_weights$ by E_sum to get W_{avg} and return it back to the main algorithm for the next federated learning round. Before returning W_{avg}, it is converted back into a list of numpy arrays. This process is in accordance with equation 3 which is used to calculate the weighted average.

4.1.2 Selective Inclusion

In selective inclusion, the idea is to once again use the evaluations in the averaging process. But instead of using every user's weights in the averaging process, we only select a subset of the users that fit the criteria to be included in the averaging process. The averaging process used here is neutral and not

weighted. The criteria to be included in the averaging process is that their evaluations are within a certain range which is considered to be acceptable. More specifically, their evaluations are at most one standard deviation worse than the mean of all the participating users. We chose this criteria because if a user's evaluations are worse than one standard deviation of the mean, the user is not doing as well as the other users. So their weights are not as valuable as the other users' weights. So we do not include them in the averaging process. The averaged weights are then calculated from weights of the users who fulfil this criteria. Naturally, this approach requires the user U_i to share their latest evaluation E_i along with their weights W_i to the central agent C for averaging. Since this is already done in algorithms 6 and 7, there are no changes required in the algorithms for this approach and we can continue to use those two algorithms here as well.

Let us take an example to illustrate how this idea works with a user U_x. The evaluation on the test data for U_x is E_x with the metric being accuracy. The mean and standard deviation for the metrics over all users U is $\bar{E}$ and σ respectively. The weights W_x of user U_x are included in the averaging process, if and only if E_x is greater than or equal to $\bar{E} - \sigma$. If the criteria of $E_x >= \bar{E} - \sigma$ is not fulfilled, then W_x is excluded from the averaging process. If U_x is the only user to be excluded from the averaging, then the averaged weights are calculated by summing up the weights of all other users and dividing them by $n - 1$. We subtract 1 because W_x is not included in the averaging process. This can be written in a generalised form as seen in equation 6 which relies on equations 4 and 5.

$$\bar{E} = \frac{\sum_{i=1}^{n} E_i}{n} \tag{4}$$

$$\sigma = \sqrt{\sum_{i=1}^{n} \frac{\left(E_i - \bar{E}\right)^2}{n}} \tag{5}$$

$$W_{avg} = \frac{\sum_{i=1}^{n} \begin{cases} W_i, & \text{if } E_i >= \left(\bar{E} - \sigma\right) \\ 0, & else \end{cases}}{\sum_{i=1}^{n} \begin{cases} 1, & \text{if } E_i >= \left(\bar{E} - \sigma\right) \\ 0, & else \end{cases}} \tag{6}$$

In the case where the evaluation metric is loss, where a lower number is better, we need to make some changes to the equation. Equation 6 caters to a metric where a higher value is better. To make it work for loss, the criteria must be changed from $E_i >= \bar{E} - \sigma$ to $E_i <= \bar{E} + \sigma$ and then it will work for a metric where a lower value is better.

The *implementation* of this idea is once again made trivial by the fact that the only change from the weighted average approach is the need for a new averaging function which conforms with equation 6. To do so, a static method implementing equation 6 is added to the the Average class. The method is given a reference to all the users objects in a dictionary data structure. With access to U_i comes access to all the data stored in U_i as well, such as the model M_i and latest evaluation E_i. We initialise a numpy array full of zeros with the same shape as the weights W_i. This is referred to as $new\_weights$ and is used to collect the sum of the weights from users who fulfil the inclusion criteria. We also maintain a counter $contributers$ to count the number of users contributing to the averaging process by fulfilling the criteria. As this approach requires the mean $\bar{E}$ and standard deviation σ of all the evaluations E from all the users U, we must first compute these values. To do so, we iterate over all the users and collect their E_i into a numpy array called E. Then we use the API that numpy provides on its arrays to calculate the $\bar{E}$ and σ. Calling the function `mean` on the numpy array of all evaluations E returns the mean value of E. Similarly, calling the function `std` on E returns the standard deviation in E. One we have all this data ready, we once again iterate over the users in a loop where we check if U_i satisfies the criteria of $E_i >= \bar{E} - \sigma$ in the case where the metric being used is accuracy. In the case of the metric being loss, we use $E_i <= \bar{E} + \sigma$. If the user satisfies the criteria, we add their weights W_i to $new\_weights$ and increase $contibuters$ by one. To get W_i, we use the `get_weights` method associated with the Keras model M_i. This returns a list of numpy arrays which we once again convert into a numpy array of numpy arrays for quick calculations and concise code. The shape of $new\_weights$ never changes in this whole process. At the end of the loop, we divide $new\_weigths$ by $contributers$ to get the averaged weights W_{avg} and return it to the main algorithm for use in subsequent federated learning rounds. Before returning W_{avg}, it is converted into a list of numpy arrays.

4.2 Peer to Peer

In this section we will explore another one of our extensions to federated learning, this time one that makes it behave in a peer-to-peer way. Previously, the users would send their weights and evaluations to a central agent C that would average their weights and send the averaged weights back to all the users. The averaging is based on the averaging methodology being utilised by C. The averaged weights are then used by the users in their models for the next round of federated learning. But in a peer-to-peer context, the averaging process takes place on every user's device itself instead of C. Every user U_i in U sends its weights to every other user in U which means that every user U_i has access to the weights from every user in U. We call this set of weights received from other users W_{others}. U_i then performs evaluations locally using its own local testing data on its own model M_i and on models initialised from every weight from the set W_{others}. This way it can emulate models that the other users have learnt. Doing so, U_i can check how relevant everyone else's models are for its own data and take appropriate action depending on the averaging methodology being used. The evaluations obtained from this process are then used in the averaging process by U_i to calculate W_{avg}. This W_{avg} is used in the next round of local learning by U_i. It is not shared with any other users. As W_{avg} is now specific to U_i, we call it $W_{avg,i}$.

We alter algorithms 4 and 5 to reflect these changes which results in algorithms 8 and 9 respectively.

Algorithm 8: User side processing

```
1  def user(set_of_weights):
2      if set_of_weights != None:
3          new_weights = local_average(set_of_weights)
4          M_i.set_weights(new_weights)
5      evaluation = user.evaluate_model()
6      user.add_pre_fit_evaluation(evaluation)
7      for e in local_epochs:
8          M_i.train()
9      evaluation = user.evaluate_model()
10     user.add_post_fit_evaluation(evaluation)
```

Algorithm 9: Peer-to-Peer Federated Learning

```
1 def federated_learning_core(model):
2     send_to_users(model)
3     set_of_weights = None
4     for round in rounds:
5         for user in users:
6             user(set_of_weights)
7         set_of_weights = [ ]
8         for user in users:
9             set_of_weights.append(user.weights)
```

Algorithm 9 is a simulation of the peer-to-peer communication that would take place in a real world scenario where the users would broadcast their weights to one another. In the simulation, the algorithm acts as a coordinator for the process and takes care of sharing every user's weights with all users by collecting every user's weights and sending the set of weights to everyone. The algorithm starts off by sending all the users in U the model M that they will be using in this process. Then, for every round of federated learning, we start off by training all the users which is illustrated in algorithm 8. In this process, if the algorithm provides a set of weights to a user U_i, U_i will calculate the averaged weights based on the local averaging methodology and then initialise its model with the newly computed averaged weights. The averaging will be discussed more in depth in the following section. If no weights are provided to the user, then the weights of the user's model are not changed. The model is then evaluated on the U_i's local test data and the evaluation results are stored in the pre_fit evaluations. After doing so, the model is trained for a number of epochs and then once again evaluated on the U_i's local test data and the evaluation results are stored in the $post\_fit$ evaluations. After the training process, all the weights from the users are collected and sent to every user in the next round of federated learning. When the next round of federated process begins, the whole process repeats once again. We do this for a set number of federated learning rounds.

For the averaging process, U_i can utilise the *weighted average* or *selective inclusion* methodology. The standard unbiased averaging, such as the one Google proposed, makes no difference in this context. Google's approach is where the weights are simply summed up and divided by the number of users

participating to get the averaged weights, as in equation 2. In a peer-to-peer context, if every user were to do carry out that process, they would all end up with the same W_{avg}. This W_{avg} would also be the same as the one provided by C in the standard central federated learning approach. This is why using Google's standard averaging federated learning in a peer-to-peer context makes no difference, as the results would the same if it was being done in a peer-to-peer context or a central context.

The *implementation* of this framework requires minimal changes with respect to the implementation of the central approach. The changes in implementation to cater to algorithms 8 and algorithms 9 will be explained here. After the `train` method is called on every U_i, we iterate over all users again and create a dictionary mapping of user ID to their weights such that U_i's ID i points to W_i. User ID i can be accessed from U_i by using the `get_id` method. We call this dictionary $id\_to\_weights$. This dictionary is then passed into the `train` method of U_i in the next round of federated learning instead of the averaged weights which is a list of numpy arrays. In the `train` method of U_i, we now also have a check to see if the weights provided are in a dictionary data structure. If so, then the peer-to-peer learning process is being used and local averaging must be performed. So the dictionary is then passed into a method which is part of the class Average to compute the averaged weights $W_{avg,i}$ for U_i. This method is specific to the averaging methodology and will be explained in the following sections. After the local averaging method returns $W_{avg,i}$, the `set_weights` method is used to set the weights of M_i before local training commences.

4.2.1 Weighted Average

In this section we discuss how federated learning with weighted averaging can be carried out in a peer-to-peer context. The generic peer-to-peer federated learning framework is made up of algorithms 8 and 9. Line 3 in algorithm 8 is where the local weighted averaging process takes place. Weighted averaging in the context of a central agent is described in Section 4.1.1. The peer-to-peer version is very similar to what is described there. The basic idea still remains the same. We wish to weigh the weights of certain users higher than the others in the averaging process. But this decision is now taken by every user U_i independently and based on evaluations made on U_i's local test data.

For example, let us take a user U_x with model M_x and weights W_x. As per the peer-to-peer framework U_x receives weights W_y and W_z from U_y and U_z respectively. The criteria for U_x's weighting of W_x, W_y and W_z in the weighted averaging process is based on the individual evaluations of models initialised with the aforementioned weights on U_x's testing data. U_x's evaluation of M_x with W_x on its local testing data is E_x. To evaluate W_y, U_x initialises M_x with W_y and evaluates it on its own testing data. We call this evaluation E_y. The same procedure is followed with W_z to obtain E_z. By U_x always performing the evaluations on its own testing data, U_x can see how relevant a certain user's weights are with regards to its own local testing data. The best evaluation is the most relevant to U_x's data and therefore the weights corresponding to that evaluation should be weighted higher than the others. It is very likely that W_x will be weighted the highest as it has been trained on U_x's training data which shares characteristics with its testing data. All the weights and evaluations are then used to calculate the weighted average as per equation 3. The changes required based on the metric choices still apply. In the case where the choice of evaluation metric is such that a lower value is better than a higher one, such as in loss, we replace E_i by its inverse $\frac{1}{E_i}$ in equation 3. If E_i is zero for such a case, then its inverse is not defined. As mentioned before, we then choose to use 10^{-6} as E_i and then use its inverse in the process. This weighted average is specific to U_x so we call it $W_{avg,x}$. Users U_y and U_z carry out the same process locally as well to get their own weighted averages, $W_{avg,y}$ and $W_{avg,z}$ respectively. Every user then sets their local models to be initialised with their respective weighted average to carry out local training for that round of federated learning. This process is repeated for every round of federated learning.

The *implementation* of this idea required us to add more static methods in the Average class. The main method that calculates the weighted average, as seen in line 3 in algorithm 8, is named `weighted_avg_personalised` and is provided with the dictionary of weights *id_to_weights* as one of its parameters. This method starts off by calling the another static method of class Average, named `eval_user_on_all`, whose aim is to evaluate models initialised by all available weights on the current user's test data. This method is provided with an object reference to the current user U_i and the dictionary of weights *id_to_weights*. It starts off by storing the original weights of M_i, W_i, in a temporary variable *original* for future use. Another dictionary is

also initialised using the keys from *id_to_weights*. We name this dictionary *id_to_evals*. Its aim is to store and map the user ID j to the evaluations of the model initialised with W_j on U_i's test data. This is done for every user's weights. It must be noted that only in this simulation do we know the user IDs that are associated with the weights. In a real world scenario these weights would be anonymous. We then iterate over all the weights W_j in *id_to_weights*. For every iteration, we set the weights of M_i to be W_j using the `set_weights` method that Keras provides for its models and then call the `evaluate` method on U_i. This, as mentioned before, evaluates M_i, which now hosts W_j for its weights, on U_i's test data and returns the evaluation E_j. E_j is essentially indicates how well W_j works on U_i's test data. We then store a mapping of j to E_j in the *id_to_evals* dictionary. After iterating over all weights and obtaining all the evaluations, we set the weights of M_i to be *original*, which resets U_i to have its original model weights. Then the dictionary *id_to_evals* is returned to `weighted_avg_personalised`.

In `weighted_avg_personalised`, we now initialise a numpy array full of zeros with the same shape as W_i and name it *new_weights*. We use this to collect the sum of the weights as we iterate over every weight W_j in *id_to_weights* once again to calculate the weighted average. We also initialise a variable *sum* to zero which is used to keep a sum of the evaluations for use later in the averaging process. For every iteration, we convert W_j to be a numpy array of numpy arrays instead of the list of numpy arrays default structure. This is done to have succinct code and efficient calculations. Then we fetch the evaluation E_j for U_j's weights from *id_to_evals* by providing j as the key. If accuracy is the metric that is being used, then we can use E_j directly. But if loss is being used, we use the inverse of E_j, $\frac{1}{E_j}$, as E_j instead. If E_i is zero in such a case, then its inverse is not defined. So as mentioned before, we choose to just use 10^{-6} as E_i and then use its inverse in the process. We add E_j to *sum* and we add $W_j \times E_j$ to *new_weights* to update the summed values in every iteration. After iterating through all the weights, we divide *new_weights* by *sum* to get the weighted average $W_{avg,i}$ for U_i, in accordance with equation 3. This $W_{avg,i}$ is converted into a list of numpy arrays and is then used by U_i to initialise its M_i using the `set_weights` for the next round of federated learning.

4.2.2 Selective Inclusion

In this section we discuss how federated learning with selective inclusion in the averaging process can be carried out in a peer-to-peer context. We will once again utilise the previously described generic peer-to-peer federated learning framework. This framework is made up of algorithms 8 and 9. Line 3 in algorithm 8 is where the local selective inclusion averaging process takes place. The idea behind this averaging methodology in the context of a central agent is described in Section 4.1.2. The peer-to-peer version is very similar to what is described there. The basic idea of excluding users from the averaging process based on the evaluation still remains. But this decision is now taken by every user U_i independently and based on evaluations made on U_i's local test data.

For example, let us take a user U_x with model M_x and weights W_x. As per the peer-to-peer framework U_x receives weights W_y and W_z from U_y and U_z respectively. The criteria for U_x to include any of the weights W_x, W_y and W_z in the averaging process is based on the individual evaluations of models initialised with the aforementioned weights on U_x's testing data. U_x's evaluation of M_x with W_x on its local testing data is E_x. To evaluate W_y, U_x initialises M_x with W_y and evaluates it on its own testing data. We call this evaluation E_y. The same procedure is followed with W_z to obtain E_z. By U_x always performing the evaluations on its own testing data, U_x can see how relevant a certain user's weights are with regards to its own local testing data. In selective inclusion, as per equation 6, the weights included in the averaging process need to have evaluations better than a threshold value. In the case where the evaluation metric is such that a higher values are better, like with accuracy, the condition to be fulfilled is $E_i >= \bar{E} - \sigma$ where $\bar{E}$ is the mean and σ is standard deviation of all evaluations E. In the the case where the evaluation metric is such that lower values is better, like with loss, the condition is $E_i <= \bar{E} + \sigma$. The weights from corresponding evaluations that satisfy the condition are included in the averaging process. It is very likely that W_x is included in the averaging process as it has been trained on U_x's training data which shares characteristics with its testing data. This average based on selective inclusion is specific to U_x, so we call it $W_{avg,x}$. Users U_y and U_z carry out the same process locally as well to get their own local averages, $W_{avg,y}$ and $W_{avg,z}$ respectively. Every user then sets the weights of their local models to be their respective averages before carrying out local

training for that round of federated learning. This process is repeated for every round of federated learning.

The *implementation* of this approach is very similar to what was previously explained in the peer-to-peer version of weighted averaging. We implement a static method in the Average class and name it `std_dev_personalised`. This method implements the selective inclusion averaging process in accordance with equation 6. To do so, the method is provided with the dictionary of weights *id_to_weights* as one of its parameters. The method's usage can be seen in line 3 of algorithm 8. This method, when invoked by user U_i, starts off by calling another static method of the Average class called `eval_user_on_all`. The aim of this method is to evaluate models initialised by all available weights on the current user's test data. As described in the previous section, this method returns a dictionary called *id_to_evals*. This dictionary stores the mapping of an arbitrary user U_j's user ID j to the evaluations of the model initialised with W_j on the current user U_i's test data. The implementation of this method is described in the previous section. The values in the key-value pairs of dictionary *id_to_evals* are the evaluations which we will refer to as E. We obtain E by calling the `values` method on the Python dictionary object and cast the returned values object into a numpy array of evaluations. Doing so allows us to calculate the mean $\bar{R}$ and standard deviation σ of E trivially by making calls to the `mean` and `std` methods respectively on the numpy array of evaluations E. After doing so, we initialise a numpy array full of zeros with the same shape as W_i and name it *new_weights*. We use this to collect the sum of the weights. A variable named *contributers* is also initialised to zero which is used to keep a track of count of users that satisfy the inclusion condition. This is used later when calculating the average. We then iterate over all the evaluations E_j stored in *id_to_evals* and check if the relevant conditions are being satisfied. In the case of the metric being accuracy, the condition is $E_j >= \bar{E} - \sigma$ and in the case of the metric being loss, we use $E_j <= \bar{E} + \sigma$. If the evaluation satisfies the condition, we add the weights corresponding to user ID j, W_j, to *new_weights* and increase *contibuters* by one. User ID j is used as the key to index into the dictionary *id_to_weights* which returns W_j, the weights of U_j. We convert W_j to be a numpy array of numpy arrays instead of the list of numpy arrays default structure. This is done to have succinct code and efficient calculations. After iterating through all the evaluations, we divide *new_weights* by *contributers* to get the averaged weights $W_{avg,i}$ for U_i. This

averaging process is in accordance with equation 6. This $W_{avg,i}$ is then converted into a list of numpy arrays and returned to the user. U_i uses $W_{avg,i}$ to initialise its M_i using the `set_weights` for the next round of federated learning.

4.3 Localised Training

In this section, we describe the idea of localised training. With localised training, the users do not share their weights at all and keep training their model on their local training data. This is equivalent to every user performing non-federated learning locally, as described in Section 3.1. Non-federated learning is one of the baselines that we use where a single model is learnt from all the available data. This approach is similar to that but where every user learns a model and the available data is only their own local data. This acts as another baseline that we can use to see how well users perform when only training with their local data.

The upside to this idea is that the parameters that the user's model learns have more relevance to the user's local data. In theory, this should make the local predictions for a user better as the locally learnt parameters are not contaminated by the parameters from other users. That being said, if a wildly different prediction is attempted on the user's model, the chances of the prediction being unreliable are very high. This is because the user does not benefit from the knowledge that the other users have learnt in their parameters, so the scope of learning is narrower. For example, in a classification problem, if a user U_i only has pictures of different breeds of dogs, then after localised training the model will be able to classify the different breeds of dogs fairly well. But if U_i asks for an image of a cat to be classified, then the system would not be able to give a reliable prediction for the cat and possibly misclassify it as a breed of dog. This is because the system has not been exposed to images other than that of dogs, so it lacks the knowledge of classifying anything that is not a dog.

The design for this idea is based off of the central federated learning framework that is laid out in algorithms 4 and 5. We change algorithm 5 such that we no longer provide user U_i with new weights from C and we change algorithm 4 to show that no new weights are being set on the user's side.

Only training of the model takes place on the user's side. As a result, we get algorithms 10 and 11.

Algorithm 10: User side processing

```
1 def user():
2     evaluation = user.evaluate_model()
3     user.add_pre_fit_evaluation(evaluation)
4     for e in local_epochs:
5         M_i.train()
6     evaluation = user.evaluate_model()
7     user.add_post_fit_evaluation(evaluation)
8     return M_i.weights
```

Algorithm 11: Local Learning in the federated framework

```
1 def federated_learning_core(model):
2     send_to_users(model)
3     for round in rounds:
4         for user in users:
5             user_weights = user()
```

The procedure still follows the idea of training all users over rounds of federated learning, but without actually sharing weights between the users. For every round of federated learning, all the users train their models for a number of local epochs. The users also keep a note of the evaluations of their models on their local test data before and after the local training process takes place. This data is called the the pre_fit and $post_fit$ evaluations of the user. This takes place for every round of federated learning. The sharing of weights does not take place.

The *implementation* of this idea is trivial. It is essentially a restriction on the central federated learning implementation that has been discussed previously in Section 3.2. As such, we will only discuss the changes that need to made to implement this idea. For every round of federated learning, we iterate over every user U_i in U and call the `train` method on the U_i object. When calling the `train` method on U_i, we pass in a `None` object instead of passing in the new averaged weight like we would in the central federated learning process. In the `train` method, we have a simple check to see if the

parameter for new weights is a `None` object, and if so, we do nothing. By doing so, the user does not set the weights of their model M_i to anything new when the method is called and the weights of the model stay the same for the next steps in the method. The next steps in the method and everything else in this implementation is the same as the implementation of the central federated learning process described in Section 3.2.

So far we have seen the theory behind several different ways in which we can perform federated learning. In the next section we look at the results from experiments conducted on two different datasets to compare the different methodologies we have described in the previous sections.

5 Experimentation

Experiments using two distinct datasets were conducted to compare the performance of all the ideas. The first dataset was based on hand gestures and the second dataset was based on images. These datasets, the experiments conducted on them and the results obtained are described in Sections 5.6 and 5.7 respectively.

To make the whole process as fair as possible, a model that performed well on a given dataset with the traditional ML approach was found emperically. Using the traditional approach, a model was found whose performance was good.

5.1 Setup

tff GPU RAM

5.2 Framework

iterate over splits or just do it. Acc used epoch*rounds = rounds

5.2.1 Design

5.2.2 Implementation

5.3 Graphing

For each dataset show the otuput we got talk about tensorflwo gpu

5.3.1 Model selection

Once the data is ready, it is essential to find a model M that performs well on the given data. The process of finding an M that fits the dataset well is an empirical one. It involves numerous experiments with different model architectures and tweaking of hyper-parameters. The model with the best metrics on the validation data is selected as M. This is a crucial step, as when we conduct experiments with the same dataset in a federated setting, the same model M will sent to every user U_i during the initialisation process. The reason behind this is to have a fair set of experiments where the model

is not specifically chosen to work well in a federated environment.

We have to then try to find a model M that performs well on the dataset. Keras provides an API to construct a model with relative ease. It allows us to construct a layer of neurons and stack them on top of each other to build an architecture suitable for our needs. The layers include, but are not limited to, the layers mentioned in Section 2.4. To find the best M, several different architectures were tested on the dataset in question. The tests included training the model on the training data and then plotting a graph of metric versus epoch to see how the metrics progress over time. The metric of choice was generally accuracy. The plots were made for both, the performance of the model on the training data and the validation data. Efforts were made to ensure that the model was not suffering from over-fitting or under-fitting. The plots were very useful in checking for that. An indication of over-fitting is that the metrics of the validation data and training data begin to diverge. In this case, the model is made a bit simpler. Techniques include reducing the number of neurons, number of layers, adding regularization or adding a dropout layer (where certain neuron outputs are ignored). A sign of under-fitting is that the metrics are too "bad", for instance the accuracy being too low. In this case, we can do the opposite of what we do in the case of over-fitting. All of these techniques can be easily implemented in Keras using the API it provides to create and edit layers.

5.3.2 Data separation

INCOMPLETE After explaining how to split the data for a given user, we need to look at how to allocate data to a specific user in the first place. Datasets can come in many shapes and forms. Some datasets are structured in csv files and some datasets are structured using a directory structure. Sometimes they come with explicit user identities, specifying what part of the dataset belongs to what user, and some come without any user identity. Here we will explain at two ways in which datasets are delivered that are most relevant to this project.

In the case where the dataset is delivered in a csv file with a column dedicated to indicate the user ID that a sample belongs to, the separation is trivial. The idea is to iterate over the csv file and based on the user IDs, separate the data. To do so, the csv file is read into a pandas DataFrame.

We can call this *df*. As the user ID column indicates what user a sample of data belongs to, filtering *df* on the user ID column is enough to separate the data. Pandas DataFrames have a filter functionality which allows us to do this with ease. First, we find the unique user IDs in the column which we can easily do using the DataFrame API. We then iterate over all the user IDs and filter *df* so it return DataFrames that contain data only for a particular user. We maintain an array of references to these DataFrames and return it for further processing to split it as in Section 5.5. Algorithm **??** shows the same in a Python based pseudocode below.

Algorithm 12: Data separation for csv files with user IDs

```
1 def separate_csv(df):
2     user_dfs = [ ]
3     # returns a list of unique user IDs
4     user_ids = df["UserID"].unique()
5     for user_id in user_ids:
6         # returns a DataFrame with data for user with ID user_id
7         df_user = df[df["UserID"] == user_id]
8         user_dfs.append(df_user)
9     return user_dfs
```

In the case where the dataset is delivered using a directory structure and no user IDs, we have some freedom in terms of the separated. We can use the idea of *simulated federated data*. This is where artificial users are created and given certain characteristics. A sample of a characteristic which we will look at later in section 5.7 is image affinity. We can create users have a higher affinity to a certain class of images more than classes, i.e., a user who has 80% of their images as cats and the other 20% are random pictures from the given dataset. The 80-20 ratio, as mentioned previously, can be changed to any other ratio we may want to experiment with because it is was artificially set by us in the first place. The directory structure indicates the classes of images, for instance, images in a directory called *cat* are labelled as *cat*.

To implement the simulated federated data idea, we

Algorithm 13: Data separation for csv files with user IDs

```
def separate_dir(dir):
    # valid extensions
    VALID_IMAGE_EXTENSIONS = [".jpg", ".jpeg", ".png"]
    user_ids = df["UserID"].unique()
    for user_id in user_ids:
        # returns a DataFrame with data for user with ID user_id
        df_user = df[df["UserID"] == user_id]
        user_dfs.append(df_user)
    return user_dfs
```

The implementation begins the same way by splitting the data as in Section 5.5 and storing it in user objects defined in Python as seen in Section **??**. But first, we must deal with the fact that the dataset provided here is a combination of every user's data in one csv file. So before using the splitting logic on it, we must first read in the dataset using pandas and assign users the data that should be local to them. The logic behind this is specific to the dataset being used, and will be discussed in Section **??** where we conduct experiments on actual datasets. But the crux of the matter is to read in the dataset and separate the data specific to every user and then apply to splits to the user's local data.

5.4 Proportional Splitting

With that in mind, we talk a little bit about the percentage of user data present in the three categories that we mentioned above. We talk about this here specific to the . There are two ways to split user based data, the naïve way and a more sophisticated way based on the idea of stratification. The naïve way is to take the whole dataset and just split it into training, validation and test data. If this were to be the case, then it is almost certain that there would be user data which would end up not being present in a given split. For example, it would lead to a situation where data from user U_i would not be present in the training data at all, but possibly take up the entirety of the test data. One could suggest that shuffling the dataset before the splitting would fix this issue, but that is not the case. It is better than the plain naïve approach described above, but data from a user U_i might still end up being under represented in either of the three categories of data that we store.

The more sophisticated approach which based on the idea of stratification addresses the issue of users being under represented in the training, validation or test data. In this approach, we split every user's data into the three categories. Then, for each categories, we take the union categorical data for every user. This union then acts as the data for the *global_user*. For example, after splitting every user's data, we will take every user's test data and union it to get a collection of test data from all the users. This collection of test data acts as the test data for the the *global_user*. With this approach we can ensure that every user is represented in accordance with the size of their dataset. This can be visualised in Figure 9.

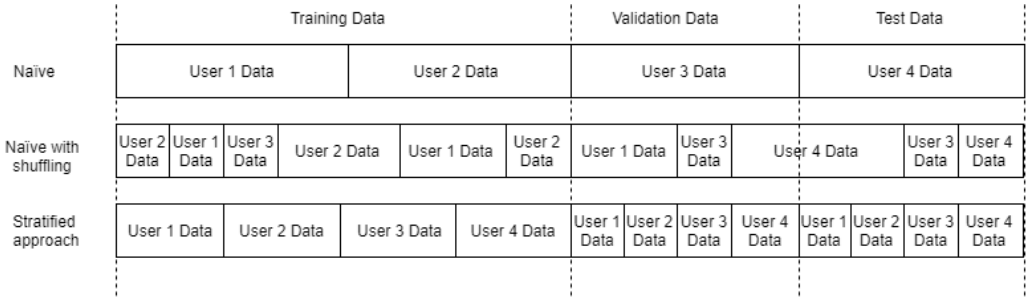

Figure 9: A visual representation of the data splitting approaches.

To store this data, we use the idea of a User object which is described in Section **??**. Algorithm 14 illustrates the stratified split idea well by using pseudocode. For every user, the data is split and added to a collection of the same type of data. After iterating through all the users, a *global_user* object as described in Section **??** is instantiated and the data stored in it. Then the $global_user$ object is returned for further usage.

Algorithm 14: Stratified data collection for the global user

```
def global_user_data_init(users):
    unioned_train = [ ]
    unioned_validation = [ ]
    unioned_test = [ ]
    for U_i in users:
        train, validation, test = split_user_data(U_i)
        unioned_train.append(train)
        unioned_validation.append(validation)
        unioned_test.append(test)
    global_user = User()
    global_user.set_train_data(unioned_train)
    global_user.set_validation_data(unioned_validation)
    global_user.set_test_data(unioned_test)
    return global_user
```

5.5 Training, validation and testing splits

As a rule of thumb, any dataset provided needs to be split up into the three parts, training, validation and testing data to avoid leakage. A common strategy was used through the project to ensure consistency amongst all experiments. This strategy is explained in this section.

The dataset provided is usually in a csv file and is read into a DataFrame using the API that pandas provides. To have flexibility, we extract the data as a numpy array from the DataFrame and work directly with the array. We split the array into three parts, training, validation and test data, where they have 60%, 20% and 20% of the data respectively. This split ratio is the industry standard when it comes to three splits like we do in this project. Every split has a specific task. The training data is used exclusively for training the model. The validation data is used to see how the model performs on unseen data that is not the test data. We can inspect this data, analyse how the model performs on it and tweak the model appropriately. We do not do the same with the test data. The purpose of having a validation set is to avoid leaking knowledge of the test set into the training process of the model. The validation set is not directly used in the training of the model. The test set is used at the very end to evaluate the final performance of the model.

To implement the splitting, context specific functions were defined each of which would be used based on the input data type. The parameters for the functions would include `full_data`, `for_user`, `val_size`, `test_size` and `seed`. The values for `test_size` and `val_size` in this project were 0.2 and 0.2 respectively, giving giving 0.6 to the training set. These fractions represent the fraction of the whole input dataset to be allocated to the three named subsets. The logic of the splitting requires the input data to first be split into training and test data on the specified split ratios of 0.8 and 0.2 respectively. Then the training data must be split into training and validation data. As the split now takes place on 80% of the input data, we need to adjust `val_size` to ensure that the correct split ratios are maintained with respect to the full input data size. To do so, `val_size` needs to be recalculated as in equation 7. The training set is then split into training and validation data based on the new `val_size`.

$$val\_size = \frac{val\_size}{1 - test\_size} \tag{7}$$

The `seed` ensures reproducibility because of the shuffling involved in the splitting method provided by scikit-learn. The parameter `for_user` is used to split the data pertaining to only a specific user in the dataset.

5.6 Gestures dataset

Since the data is already a tf.data.Dataset, preprocessing can be accomplished using Dataset transformations. Here, we flatten the 28x28 images into 784-element arrays, shuffle the individual examples, organize them into batches, and renames the features from pixels and label to x and y for use with Keras. We also throw in a repeat over the data set to run several epochs.

talk about hdf5 scaling artificial users

5.6.1 Description

5.6.2 Model Selection

5.6.3 Results

Local Training
TensorFlow Federated

Standard Federated Learning
Non-Federated Learning
P2P Weighted Average
P2P Selective Inclusion
Central Selective Inclusion
Central Weighted Average
found issues so moved on

5.6.4 Testing user weights on global data

5.7 Image dataset

5.7.1 Description

5.7.2 Model Selection

5.7.3 Results

Local Training
TensorFlow Federated
Standard Federated Learning
Non-Federated Learning
P2P Weighted Average
P2P Selective Inclusion
Central Selective Inclusion
Central Weighted Average

5.7.4 Testing user weights on global data

Can just saw 6 other graphs were xyz for whatever reason

6 Conclusions and future work

can use "i" in here

early stopping could have implemented the exact version of google with deltas being sent secure aggregation could have been explored

Bibliography

[1] F. Bre, J. M. Gimenez, and V. D. Fachinotti, "Prediction of wind pressure coefficients on building surfaces using artificial neural networks," *Energy and Buildings*, vol. 158, pp. 1429–1441, 2018.

[2] F. Chollet, *Deep Learning with Python*, 1st. USA: Manning Publications Co., 2017, ISBN: 1617294438.

[3] JustinB, *Introduction to perceptron: Neural network*, 2017. [Online]. Available: `https://blog.knoldus.com/introduction-to-perceptron-neural-network/`.

[4] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, *Federated optimization: Distributed machine learning for on-device intelligence*, 2016. arXiv: `1610.02527 [cs.LG]`.

[5] A. Moodley, "Language identification with decision trees: Identification of individual words in the south african languages," PhD thesis, Jan. 2016. DOI: `10.13140/RG.2.2.25539.81445`.

[6] M. A. Nielsen, "Neural networks and deep learning," 2015.

[7] A. Segal, A. Marcedone, B. Kreuter, D. Ramage, H. B. McMahan, K. Seth, K. Bonawitz, S. Patel, and V. Ivanov, "Practical secure aggregation for privacy-preserving machine learning," in *CCS*, 2017.