Federated Machine Learning

A research project

Karan Samani
1161081087
Final Year Project BSc. Computer Science (Hons.)
Supervisor: Dr. Derek Bridge
Department of Computer Science
University College Cork
March 14, 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in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.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.

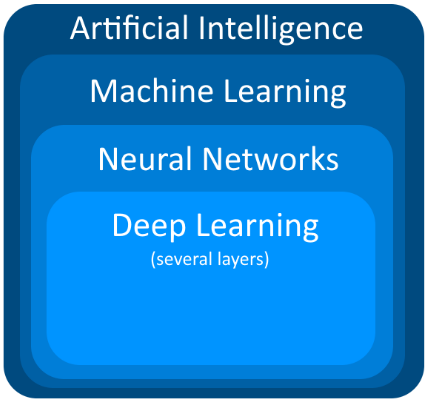

Figure 1: High level visual representation of what fields there are in the study of AI.

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 Outline of the report

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 2.7 we will see how Google describe federated learning before discussing the design and implementation in Section 3.3. 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.

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

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.

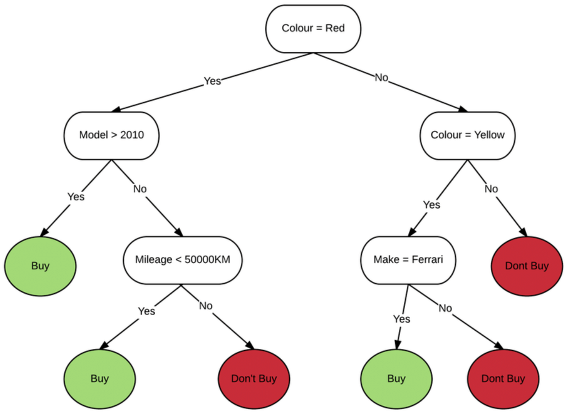

Figure 2: A very simple decision tree [5].

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.

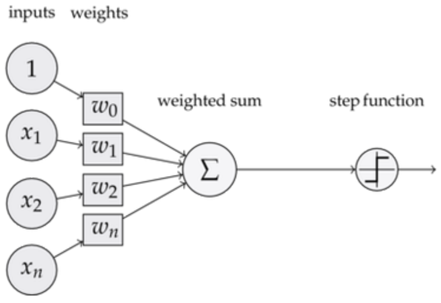

Figure 3: Major components of a neuron [3].

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.

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

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.

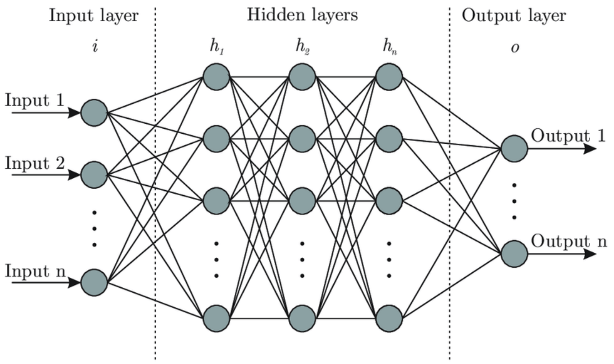

Figure 4: Basic dense neural network [1]

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.

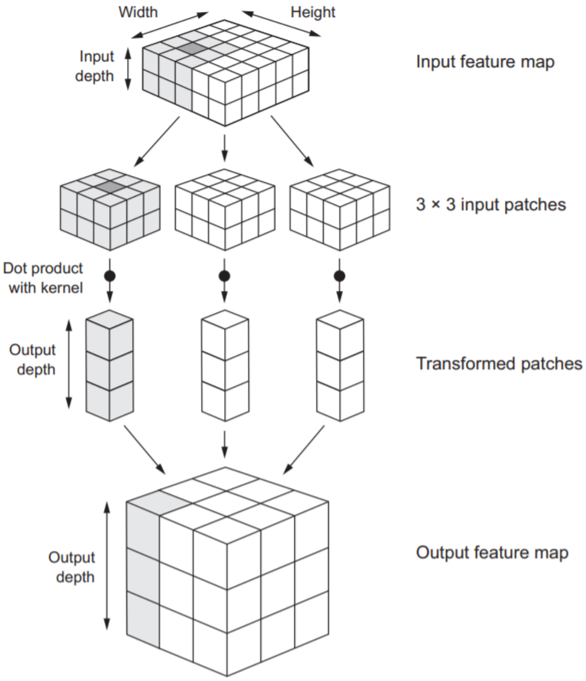

Figure 5: How a convolution works [2].

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 Differential Privacy (MIGHT OMIT THIS)

linkage attacks, netflix and imdb data to identify only good for large dataset, else bad complex easier to do real data and anonymise

deidentify data (outliers, fields)

want to decouple learning about an individual and learning about the population

analysis output is not dependant on a particular individual and will be the sameif they are not incuded

adjacent dataset n+1 and n sized dataset, algo m

2.7 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, 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 such that user U_i receives model 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

```
def user(new_weights):
    if new_weights != None:
        $M_i$.set_weights(new_weights)
    for e in local_epochs:
        $M_i$.train()
    $M_i$.send_weights_to_server()
```

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 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 server(weights):
2     $w_{avg}$ = average(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 product can be pretty close to the traditional ML approach. Although, it must be noted that depending on the application, it may not be close or some changes may be required. 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. After that, for every round r_i 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.

Algorithm 3: Federated Learning Core overall algorithm

```
def federated_learning_core($model$):
    $C$.send_to_users(model)
    new_weights = model.weights
    for $r_i$ in $rounds$:
        for $U_i$ in $users$:
            $U_i$(new_weights)
        new_weights = server(weights)
```

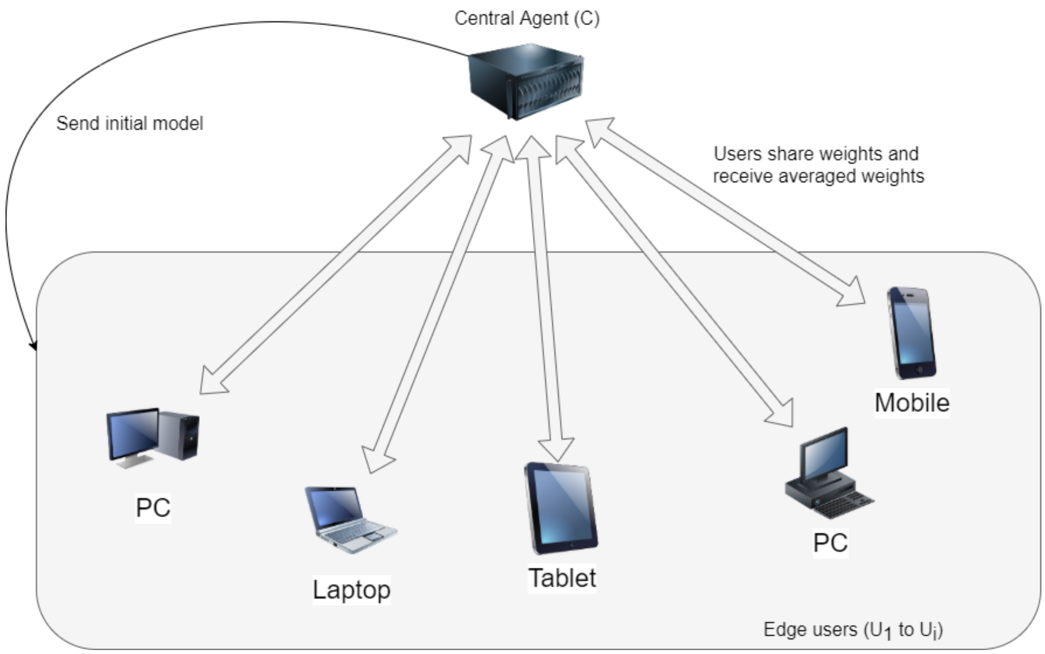
Central Agent (C)
Send initial model
Users share weights and receive averaged weights
PC
Laptop
Tablet
PC
Mobile
Edge users (U_1 to U_i)

Figure 6: Federated learning.

2.7.1 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.

2.7.2 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 (5.3. 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

The language of choice for this project was Python. The biggest reason for choosing Python was because of the Keras library. Keras is a Python deep learning library which (in this project) uses with TensorFlow as its back-end. Keras provides a user friendly API to use Tensorflow to build neural networks in Python. We chose to use TensorFlow as Keras' back-end because we will be using TensorFlow Federated (TFF) in Section 3.4 as a sanity check to confirm that our implementation of federated learning matches that of Google. TFF provides a framework using which people can build a federated learning system. Matplotlib is a Python library used for plotting graphs and it was used extensively in this project. 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 libraries which were used for data representation throughout the project.

In this section we will go through the design and implementation aspect of the basic approaches before moving on to describing the extended ideas explored in this project.

3.1 Universal strategies

Listed here are the techniques and strategies used throughout the project.

3.1.1 Data representation

As this is a project about exploring different ideas of federated learning, so there was no need to spend time on the networking aspect of federated learning by having networked users. Instead, 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.

insert class diagram

A User class was defined and its instances were used to represent user U_i. These instances of the are called user objects. A user object is used to store all data relevant to a particular U_i. We refer to this user object as UO_i.

The data stored on UO_i includes the training, validation and test data, the model of the neural network and 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 (faster) and also 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 was stored as an attribute of UO_i along with the metrics being stored in a list data structure as an attribute in the object instance as well.

3.1.2 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 the 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 3. The training set is then split into training and validation data based on the new `val_size`.

$$\texttt{val\_size} = \frac{\texttt{val\_size}}{\texttt{1-test\_size}} \tag{3}$$

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.

3.2 Global Approach

Before talking about how the federated approach is implemented, we first set in place the traditional machine learning approach. We will refer to the model created by this approach as the *global_model*, which assumes that we have access to data from every user. We name the imaginary user with this model and all the data as the *global_user*. We use this model's performance as *the* benchmark to beat. If any federated approach can be close to this approach, then we can say that we can preserve privacy whilst maintaining a similar level of performance.

3.2.1 Design

This approach is quite straight forward on its own, but the decisions taken here affect the federated approaches as well. Most notably with the choice of the model used in this approach. The high level view into the design process for the approach involved finding a way to split the data, selecting a model, to store the data, to pass the data to the model for training and then recording the metrics to analyse the performance.

There are two ways to split the data, the naïve way and a more sophisticated way based on the idea of stratification. The naïve way was to take the dataset as a whole, and split it based on the given ratios to get training, validation and test data. The idea of having these three splits is explained in Section 3.1.2. The dataset provided would be an ordered set of user specific data. If this dataset was split without shuffling, then almost certainly data from the last recorded user U_n would not be present in the training data, but it would be in the test data. Similarly, validation data might be data from just one particular user U_i and nothing else. One could suggest that shuffling the dataset before the splitting would fix this, but that is not the case. It would be 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.

POSSIBLE DIAGRAM FOR THIS?

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 the parts. Then, for every part, we take the union over every user. This union then acts as the data for the *global_user*. For instance, 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. To store this data, we use the idea of a User object which is described in Section 3.1.1.

Algorithm 4 illustrates this 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 3.1.1 is instantiated and the data stored in it. Then the $global_user$ object is returned for further usage.

Algorithm 4: 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.add(train)
        unioned_validation.add(validation)
        unioned_test.add(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
```

Once the data is ready, it is essential to find a model M that performs well on the given data. This is done by empirically finding the best model for the given data. The process involves a lot of 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. After the model is found, data can be fed into it to train the model. The metrics during the training process are recorded so that we can analyse the performance of the model post training.

3.2.2 Implementation

The implementation follows the same fundamental structure seen in the design. We have looked at the process for the approach involved finding a way to split the data, selecting a model, to store the data, to pass the data to the model for training and then recording the metrics to analyse the performance. Now we look at how these ideas are implemented in Python.

Using the process described in Section 3.1.2, the dataset is read in and split into training, validation and test data. The splitting is done with the stratified splitting approach as described in algorithm 4. This split data is stored in User objects as described in Section 3.1.1. The user object for an arbitrary user U_i is represented as UO_i.

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.

After M is selected, passing in the data for training is trivial. A method called `fit` associated to the model object is used for the training of the model. The arguments passed in are the training data, validation data, epochs and the batch size. The training data is what the model trains on and the validation data is used for performance review. Epochs is the number which specifies the number of times the training process iterates over the whole 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.

The model, whilst training, maintains a history of the metrics. After every epoch, the model is evaluated on the training and validation data, and the result is stored in a dictionary in the model object. This can be accessed by us and used to plot the aforementioned graphs in matplotlib.

STOP HERE

Implemenation
Heres the code using keras
Conv and non conv
Say how keras gets weight by making refernce to algorithms

3.3 Federated model

With the global model done and a benchmark in place, we can discuss the federated model. It is exactly explained before in Section **??**

Algorithm 5: Process taking place on

```
$M_i$.set_weights(new_weights)
store($M_i$.evaluate())
for $e$ in $local\_epochs$:
    $M_i$.train()
store($M_i$.evaluate())
$M_i$.send_weights_to_server()
```

3.3.1 Design

There is an obvious need to simulate users and a central agent in this approach and to keep data localised to those agents. This is where the idea of using a User object as described in Section 3.1.1 helps a lot to keep the data localised to a particular user.

3.3.2 Implementation

3.4 Google's Tensorflow version

Sanity check was close enough to mine

3.4.1 Design

3.4.2 Implementation

4 Extended Ideas

4.1 Central Server

4.1.1 Weighted Average

4.1.2 Excluding based on std dev

4.1.3 Local only

4.2 Peer to Peer

4.2.1 Weighted Average

4.2.2 Excluding based on std dev

4.2.3 Local only

5 Evaluation

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 5.2 and 5.3 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.
• Comoing up with decent model • Traditional ml results

5.1 Designing the framework

The testing framework was fairly simple, it involved iterating over the different scenarios and recording the results. There were

5.2 Gestures dataset

5.2.1 Description

found issues so moved on

5.3 Image dataset

5.3.1 Description

h5 files scaling artificial users

5.4 Graphing

For each dataset show the otuput we got

5.5 Testing user weights on global data

splitting of data

Analysis
Design
Implementation
Evaluation
Conclusions

third person
present or past tense

design talks about apckages and stuff. convince them its good describe test set Can just saw 6 other graphs were xyz for whatever reason Give a reason to why im shoeing something

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

7 notes

Not written in the passive Name the agent Describe the system Give it a name User Name parts of the system Edge agent, central agent Present and past

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. (2017). 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.