Abstract

The introduction of federated learning has led to the availability of using a distributed dataset containing sensitive information about the user whilst keeping it private. This increase in transmission of data between the central server and edge devices has led to a strain on the network. Previous research focused on the use of compression techniques and optimising hyper-parameter of federated learning systems but at the substantial cost of its performance. In my project, I will put an emphasis in the design of an algorithm that will select devices that contribute the most to the global model's convergence whilst rejecting sub-standard devices and so will alleviate network resources. The findings show that significant reduction of data has been achieved whilst minimising the degradation of the model's performance.

Contents

1 Introduction

1.1 Motivations and Objectives

In the past decade, there has been rapid technological growth and humanity has entered in a new era in world history. The introduction of mobile devices to the general population to the widespread use of social media applications has led to an incredible wealth of data being produced [1]. With such a large set of data, the only feasible way to analyse it is through implementing a machine learning algorithm [2] Social media's goal is to use this data analysis to retain users for longer and offer their targeted advertisement services for commercial use in the new age.

Recent controversies include Facebook's handing of personally identifiable information of 87 million people to Cambridge Analytica, which used this data specifically to influence targeted consumers and voters. This sets a bad precedent, raising questions regarding the integrity of an election result due to suspicions of malicious actors and the safety of the individual's rights [3]. The user's data privacy must be considered when conducting machine learning.

Furthermore, the typical implementation of machine learning is by gathering this data to a central cloud server and executing the process centrally. This is unsustainable and unfeasible because the data can be so large that machine learning operates very slowly in this state[2].

The use of federated learning is a good alternative. Instead of utilising a large central data set, it utilises smaller data sets where it stores and does the training process locally. This allows for lower communication overhead as it only sends the training result to the server and results in the process being completed faster due to parallel computation [4]. This leads to the central server aggregating the local training result that it has received to improve the global model. The updated global model will then be shared to local devices. Federate learning helps keeps user information private as the user's data is not leaving their device, and the training results are aggregated with other training results from different devices so that the global model will represent a variety of data sets.

However, the sheer scale of data being produced puts considerable pressure on the wireless communication channels [5]. The inherent resource constraints in networks don't allow for the mass inclusion of edge devices in federated learning due to the limited bandwidth assigned to data transmission when uploading happens simultaneously. As we have seen data gathering increase exponentially in the past, I believe this trend will continue and applying it to the current network design will only increase computation overhead, so it takes much longer to complete the training models. In addition, there will be more of a demand to create large scale models to solve more complex problems, which federated learning won't be able to achieve [6].

This paper aims to build an image classifier using PySyft, design an algorithm that will focus on the aspects of client selection in federated learning to reduce the communication overhead in a network and test the scalability of the improved model.

2 Background

2.1 Machine Learning

The basic premise of machine learning is to take an extensive data set with a specific quality that helps efficiently train a data set to optimise a performance metric in the algorithm.
Different types of learning are employed; however, the most common is supervised learning. Its popularity is due to its applicability in modern problems, particularly when you have a small collection of data that a relationship between the input and respective labelled output, such as in-text classification. Using this specific type of data when training makes it easier to help identify and compare the accuracy of the predicted result with the labelled correct output using the same data. This represents the idea of what's called the function approximation problem [7], [8].

Using unsupervised learning is another type of method in use specifically for data that hasn't been labelled, and the goal is to find any patterns that lead to any insight not known before. Furthermore, in reinforcement learning, there is a combination with the other two types of learning as the set data used does not have labelled outputs, but there lies a vague direction in whether the intended is right or wrong. If it is correct, the model is rewarded, and the algorithm optimises the desired behaviour [7].

2.2 Deep Learning Architecture

Supervised learning is heavily involved in the use of deep learning architecture, this includes the use of multiple layers which the first is an input layer, the next is the usage of one or more hidden layers and finally an output layer. Each layer is made up of a set number of neurons and they are all interconnected with neurons from layers that are next to them these interconnections are represented by matrices called weights. The weights are changeable to achieve the optimisation of an algorithm and so are vital to describe how important certain inputs are. The use of multiple layers when performing machine learning helps to enlarge the scope of analysing the raw input data which different layers can be assigned to optimise the different types of parameters [9].

2.3 Convolutional Neural Network

A type of deep learning architecture is a convolutional neural network which consists of many convolutional, pooling and fully-connected layers. It is primarily used to be trained to interpret images as it is easily able to impose a grid pattern over them. This makes it one of the reasons why this CNN are so well suited to using images as data when feeding it through a convolutional layer. By turning images into pixel values, we can represent a subsection as a 3x3 matrix. We apply a scalar product between this with a 3x3 kernel matrix, giving us a new representation of this small subsection of the data. This is defined as convolving which is applied to the rest of the images subsections and formatted to its relative position of the feature map [10], [11].

Each of the convolutional layers in a CNN are assigned a different purpose in interpreting an image and each of the kernels plays a significant role in this. A kernel acts as a filter in the convolutional layer where the value assigned to the kernel matrix serves the purpose of that layer's optimisation such as some layers prioritising to distinguish the different edges on an image [10], [11].

2.4 Pooling Layer

The convolutional layers output is often preceded by the pooling layer. With so many learnable parameters from the feature, it would require a lot of time to train this type of model. To make the computation more efficient, the model's complexity must be made simpler. Pooling is done by reducing the size of the feature map and so there will be a reduction in the model's parameters [11].

There are many ways to conduct the pooling method and the most popular is max pooling. Using the usual 2x2 filter window with a stride of 2, this would be applied to the first subsection of the feature map and the max value in this window will be included in its positional output. This iterated over the entire feature map. Another method is average pooling where instead of using the highest value in the filter window, the average of the value inside the 2x2 matrix is used and included in the output [10].

2.5 Fully-connected Layer

The last layer to feature is the fully connected layer which directly connects the input and output of the layers next to it. Its role is to use all the information that has been passed through the convolutional and pooling layers and helps to categorise the prediction to the output layer[11].

2.6 Activation Function

Each layer is made up of neurons and what follows these layers are different types of activation functions. Their behaviour is inherently nonlinear and when a neuron takes the sum of its weights from its many inputs from the previous layer, the outputs results will be affected by this non-linearity due to the activation function [12]. Introducing non-linearity into the model is important as this allows to include more layers in the model which means that is more suitable to train on complex problems and the ability to train on data that is non-linear itself.

In CNNs the most popular activation function to use in between the layers is the rectified linear unit (*ReLU*) and this is because training models are much quicker than the use of other activation functions [12]. This can be attributed to the simplistic calculation it performs when comparing it to others such as tanh, sigmoid and swish. Their calculations include the use of exponentials and divisions, whereas *ReLU* doesn't which saves on computation.

Tanh:
$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad (2.1)$$

Sigmoid:
$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

ReLU:
$$f(x) = max\,(0, x) \quad (2.3)$$

The activation functions that follow the last fully-connected layer are different to those being used in the other layers of the model and depending on the goal of the model, a certain activation function will be well suited for it [10], [12]. CNNs are used in image classifications so the use of a *sigmoid* or a $softmax$ activation function would be ideal to represent the probabilities.

2.7 Loss Function

When the data has passed throughout the model and reached the output layer, the predicted result of the model is compared with the actual labelled result. The comparison is used to find how much error there is between the two and so this will help towards indicating the loss function. There are many ways to calculate the loss function but when considering implementing classification, cross-entropy is ideal. It is well suited in combination with the $softmax$ activation function as it outputs the probabilities of the last layer, this helps easily calculate the error and the loss shows a logarithmic behaviour [12]. This is when a steady improving prediction will lead to a slower decreasing loss value relative to a worsening prediction in which there will be a rapid increase in loss.

2.8 Training The Model

The loss function plays an important role in the training of a model to accurately perform a task. The model is aiming to get the lowest loss possible and the results from the output layer help to determine what new value to update the weights and biases to achieve this. An optimisation algorithm executes this updating of the learnable parameters. Many models are trained using gradient descent, where they calculate the gradient of an objective function with respect to the model's learnable parameters. Including the gradient in the calculation helps to distinguish between the important weights of a model and gives them more value [9].

Backpropagation efficiently computes the gradients in the model by first starting on the layer preceding the output and working through the model to just after the input layer. In combination with stochastic gradient descent, it makes the optimisation algorithm much more efficient. Gradient descent algorithm computes using a full data set whereas stochastic gradient descent subsection of the data set and so the computation is much more efficient but to the cost of its convergence [9].

2.9 Federated Learning

The main aspect of federated learning is the ability to gain access to the use of an enormous set of data that is found in local devices whilst keeping the raw data private. This allows for complex models to be trained due to the availability and variety of the data set. However, due to the decentralised nature of federated learning, clients exhibit a varied distribution of different properties such as the quality and quantity of their data also their ability to upload the model results to a communication network [13],[14].

There are different types of federated learning that are used according to the type of dataset. When a dataset has the same classifying labels but are in a different sample category, horizontal federated learning is used especially when training a supervised learning task as their labels are already included [15]. Another type is called vertical federated learning, the dataset has different classifying labels but are in the same category as each other [15].

Federated Learning can be adapted to what type of devices that can participate in the training. When using a smaller set of participants and are able to train in all iterations we identify this with cross-silo federate learning [16]. These participants are usually servers that hold the company's data. Cross-device federated learning will use edge devices such smart phones in their training process and deal with a large population participating [16]. This type of system is mostly performed asynchronously, as their lies a lot of uncertainties. These include varying levels of hardware specification which means that performing backpropagation could take a lot longer on substandard device also their availability is not guaranteed as the could device running out of power or move into a tunnel were it will not be able to upload its parameters [17]. If this was performed synchronously then it would very fragile to the occurrence of stragglers, this would keep other successful participant waiting and reducing the efficiency of the system [17].

2.10 Client Selection in Federated Learning

With multiple uploads occurring from a large available set of clients, this would increase the communication overhead of the network. Choosing a set population of which client models to upload their parameters to the central server for aggregation, helps allocate the limited bandwidth to the clients that have a more important model.

In their paper Goetz et al.[18] focused on selecting clients that had the most important data to contribute to the performance of the model however they incorporated active learning methods. To identify the valuable data, the loss function was computed and turned into a sampled probability distribution which this will be used to select the best clients. They did achieve a reduction in the number of iterations that model performed however it performed a basic binary classification task.

In another paper, Nishio et al.[19] focused on how to make cross-device federated learning viable in cellular networks. They do this by requesting specific information from a random set of clients which this will be used to estimate the time needed for a model to train and upload their parameters. Any devices that take longer than this are reject. Results showed that by incorporating more clients, they achieved better performing model in a shorter time compared to the original framework, but their model used a simple deep learning architecture.

2.11 Predecessor Research Work

The work of my predecessor focused on implementing a federated learning system to a network mechanism called QUIC. The aim was to leverage the functionalities and properties of QUIC to reduce the communication overhead in a network. The design focused more on the multiplexing aspect which aimed at increasing the efficiency of its message delivery. His solution didn't find conclusive evidence to suggest his design was better than the original. This has led me to suggest that more research should be placed on selecting fewer clients to reduce the strain on the network.

3 Methodology

I will be explaining the process of implementing the design of the synchronous federated learning model and my client selection algorithm. I will be calling the model that doesn't include my designed algorithm as the vanilla model as this is what should represent a normal configuration of a federated learning model.

3.1 Prerequisites

At the beginning of this project, I aimed to use build upon the work of my predecessor by utilising other properties in QUIC together with federated learning however the package responsible for implementing the QUIC transport layer protocol has since been deprecated.

PySyft is an ideal network mechanism replacement as it can recreate the behaviour of federated learning whilst being a hooked extension of PyTorch, a deep learning architecture [20]. Most companies such as Facebook use PyTorch as their default AI framework. By developing my model using PyTorch, I can be able to understand how my client-selection algorithm can be tested practically.

3.2 Nature Of PySyft

PySyfts augmentation of Pytorch helps to include extra features that enable workers to communicate with each other and any changes that occur to the received data are called SyftTensors. Any history of operation conducted on the data is chained together to form a structure in which the operation is first conducted on the PyTorch tensor and propagated through the SyftTensors.[20]

SyftTensors are split into two categories called LocalTensor and PointerTensor. The LocalTensor is created when a TorchTensor is generated and is used to relay the Pytorch command from the Pysyft input command. This allows the design of only including these two tensors to be looped together by using the torch tensor to hold just the data, this makes computation much more efficient.[20]

The other type, the PointerTensor, is generated when an initialised remote worker receives data. The PointerTensor doesn't hold any of the data but just identifies where and what worker has the data [20]. As a result, this makes it easier to interact with a remote machine using PyTorch commands.

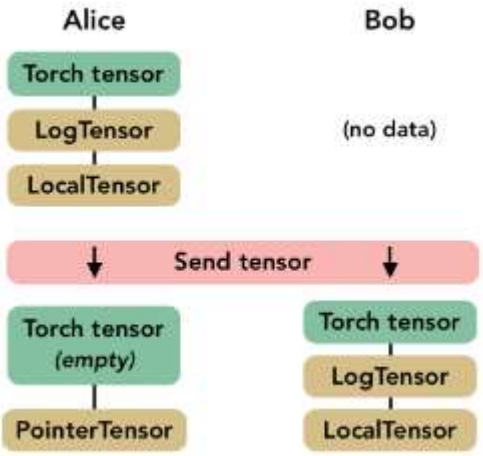

Figure 3.1: Operation of a PointerTensor [17]

3.3 Design Of The Vanilla Model

The first step is to import the PySyft and PyTorch package into python and hook it on to Pytorch. This will override the methods used on PyTorch tensors to allow tensors to be sent between workers and not doing this will not allow for it to be decentralised. This leads to the ability to include multiple remote machines in our model by representing them as a virtual worker and by storing them in a dictionary it will make it easy to call them in the future.

```
hook = sy.TorchHook(torch)
clients = dict()
for i in range(10):
  clients[i] = sy.VirtualWorker(hook,id=F"client_{i}")
```

Then we define the hyperparameters and convolutional neural network. I will be including the $ReLU$ activation in between the hidden layers of the CNN and a max-pooling layer to make the computation more efficient however the pooling layer is set to a kernel size of two to not overly reduce the model's parameters. Since we will be using this for an image classification task, I placed a $softmax$ activation function after the last fully-connected layer.

Another functionality that can be implemented is the combination of . $federate$ and . $FederatedDataLoader$. This turns the training dataset into a federated dataset where it is divided and distributed to the number of virtual workers. Therefore, this data has now been assigned PointerTensors to different locations and now it can be callable.

```
federated_train_loader = sy.FederatedDataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ]))
    .federate(list(clients.values())),
    batch_size=args['batch_size'], shuffle=True)

test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=False, transform=transforms.Com
pose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args['test_batch_size'], shuffle=True)
```

To define the train function we must first state to the model that its training and to do this is by using $model.train()$. The federated dataset is then iterated so each client can participate in the training. When communicating with remote machines, it's important to identify the PointerTensor. For this, we assign the names data and target to their respective PointerTensors.

Pysyfts provides PointerTensors to give their location by using . $location$. This can be used to locate remote machines as we know that the metadata for the PointTensor data in the virtual worker we can use this find its location. This gives the ability to send the global modal to the remote machine and can train it from there.

Before starting to train the model, all gradients must be reset to zero. This is due to the way Pytorch handles backpropagation, resulting in the accumulation of gradients in buffers []. When training a new model is important to update the parameters with correct values so it converges toward the global minimum.

Everything we need to start training is already at the remote location and connected with their respective PointerTensors. We can conduct a forwards pass on the model using $model(data)$ to get the predicted result and since target (actual label of the dataset) data is also within the remote machine, the loss can also

be calculated. Specifically, the negative log-likelihood loss as it works well in conjunction with the output of $log\_softmax$. Backpropagation can be executed by calling . $backward$ on the loss and computes the gradients which the parameters will be updated by calling $optimiser.step$. Finally, the model with its optimised parameters is ready to be sent back to the global model (local worker or central server) by calling the PointerTensor model in conjunction with . get. This will remove the model from that virtual worker and no dataset has left the remote machine.

```
model.train()
for batch_idx, (data, target) in enumerate(train_loader):
   if does_it_pass_the_decision_tree:
       model = model.send(data.location)

       data, target = data.to(device), target.to(device)

       optimizer.zero_grad()

       output = model(data)

       loss = F.nll_loss(output, target)

       loss.backward()
       optimizer.step()

       model.get()
```

To define the test function we call $model.eval$ to set the model to perform validation. The main aspect of the test function is that we are calculating the loss and accuracy of the global model locally.

```
model.eval()
test_loss = 0
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)
        # add losses together
        test_loss += F.nll_loss(output, target, reduction='sum').item()


        # get the index of the max probability class
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)
```

3.4 Client Selection Algorithm

The general aim of my algorithm is to allow remote models with acceptable results to be uploaded to the global model whilst refusing to upload remote models with unsatisfactory results. This way finite network resources will not be assigned to unimportant models but to the ones that will contribute the most to the convergence of the global model. The three most important properties that I will be measuring are the accuracy, loss and weights with respect to the model.

During the first few iterations of training the model, these parameters will be measured and recorded so their averages can be computed. The accuracy and loss averages will serve as a benchmark on whether the next model will achieve a higher result and if so, a score will be recorded in favour of uploading the model.

The weighted parameter will be averaged differently. I will be taking the tensor from the last layer and calculating the mean of it. This is why I will be using the most informative layer and building a unique benchmark system around it [21].

After a few iterations have passed, this total average of past measurements is changed to a windowed average which will only calculate the average of the three latest results. However, the weighted score for passing this benchmark will have changed for the accuracy and loss parameters. The weight parameter will be valued the same.

As I am measuring three different variables, assigning a score when they have passed or failed to surpass each benchmark will help to evaluate the importance of a model. When the Pass score is higher than the Fail score, the global model is sent to the remote machine therefore will be able to send its updated model back. If the Fail score is higher than the Pass score, the global model is not sent.

When the remote machine has been accepted by the algorithm, the model's learning parameters such as its weights and biases will be saved for future use, and this will be represented as the best parameters for the global model so far. After this, the train function will be executed as normal. However, when a client is rejected by the algorithm, the train function is skipped, and the saved global model parameters are sent in an attempt to correct the model to put it back in the right direction for convergence.

The algorithm will still record any measurements made by the rejected model and still be in the benchmark calculations. This will help the global model to learn in a much more flexible way and will stop it from getting stuck at local maximums. Not incorporating these measurements, will lead the training to become more fragile as multiple poor models can trap the training to a local maximum. Even though some poor

models will be uploaded to the global model, it will help continue to improve and reject possible future models

4 Results and Analysis

This section will show the results of the improved model (the vanilla model that incorporates the client selection algorithm) and the vanilla model. I will be comparing these two models through the lens of their accuracy, loss and number of bytes sent. The vanilla model and the improved model are both represented in orange and blue respectively.

4.1 Accuracy

Highest accuracy achieved on the improved model:

- 10 virtual workers: 96.24%
- 50 virtual workers: 96.02%
- 100 virtual workers: 96.88%

Highest accuracy achieved on the vanilla model:

- 10 virtual workers: 98.09%
- 50 virtual workers: 98.03%
- 100 virtual workers: 98.08%

The first result we see from this is that the vanilla model performs 2% better on average compared to the improved model. In the figures below, we see that there is much more volatility in the improved model which could be a result of the algorithm leading the model to reach a local maximum a lot more often than the vanilla model. Furthermore, the more virtual workers can participate in the model, we see the fluctuations of the training become much more intense also we see that the improved model does follow a similar trend to that of the vanilla model however it seems to trail behind in the first 1500 iterations but as it goes through more iterations, it seems that the improved model slowly improves its accuracy.

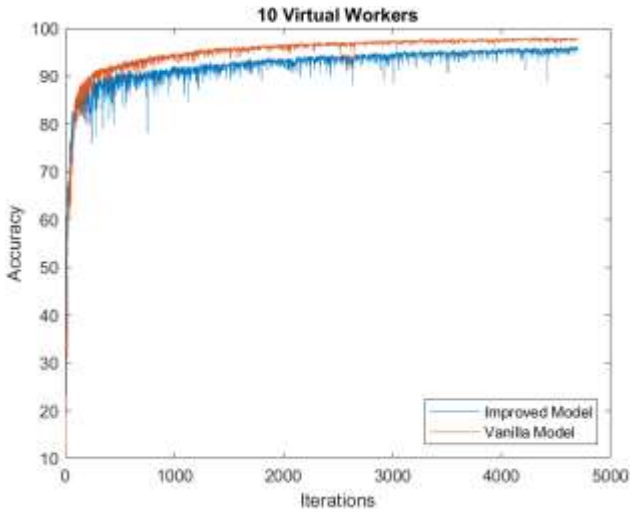

Figure 5.1

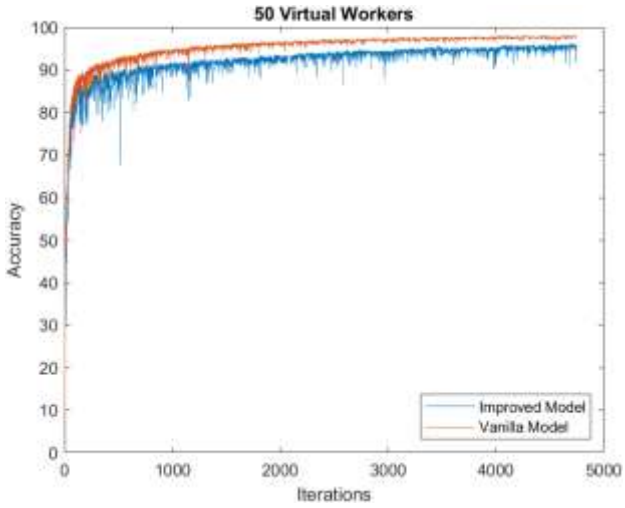

Figure 5.2

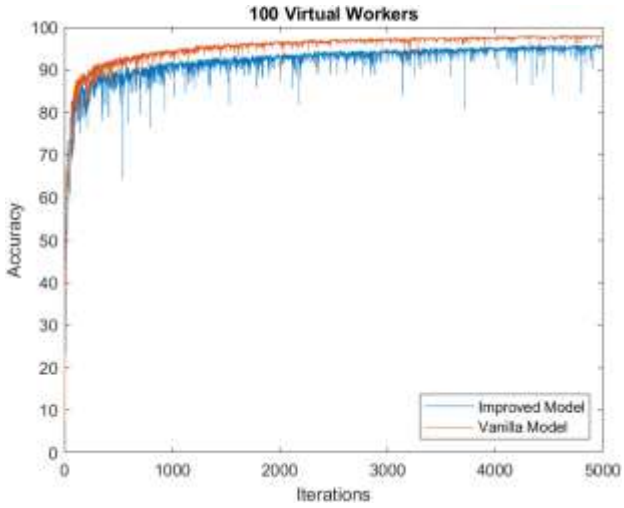

Figure 5.3

4.2 Loss

Lowest loss achieved on the improved model:

- 10 virtual workers: 0.1230
- 50 virtual workers: 0.1355
- 100 virtual workers: 0.1369

Lowest loss achieved on the vanilla model:

- 10 virtual workers: 0.0614
- 50 virtual workers: 0.0620
- 100 virtual workers: 0.0603

The vanilla model performs better by almost double relative to the improved model, but it does show that the improved model can imitate a similar behaviour to the original, just not so accurately. The improved model shares a lot of behaviours with what it was in achieving in the accuracy part. We see that the loss fluctuates a lot more in the improved model compared to that of the vanilla model and the intensity of the fluctuation increases as you increase the number of virtual workers.

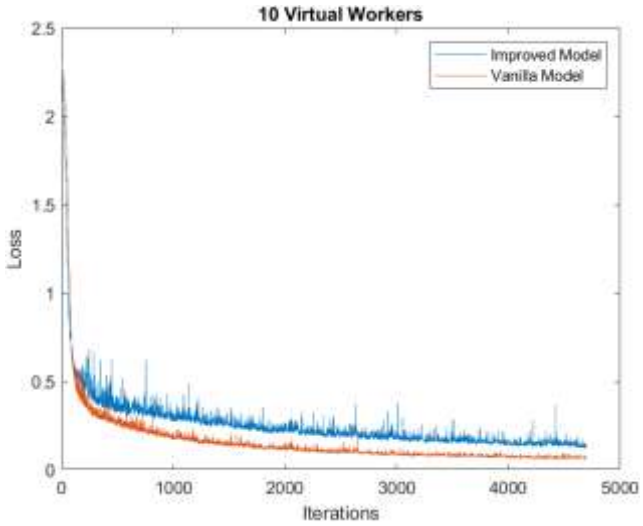

Figure 5.4

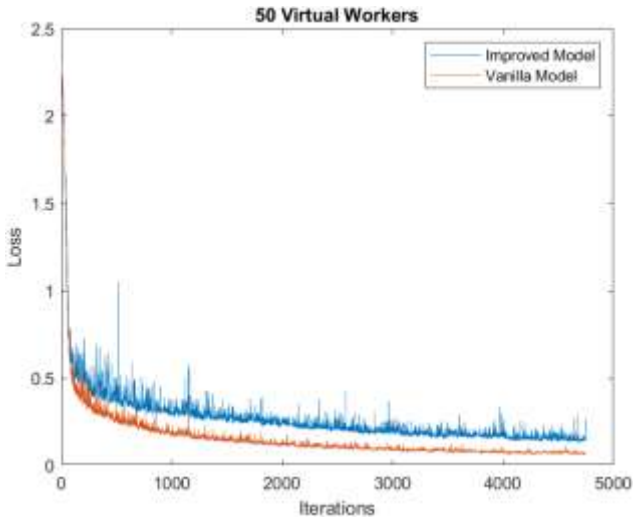

Figure 5.5

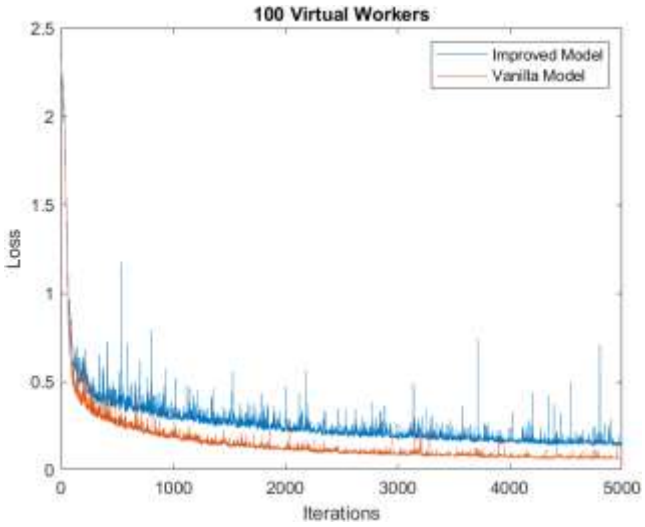

Figure *5.6*

4.3 Number Of Bytes Sent

Total bytes uploaded to the global model on the improved model:

- 10 virtual workers: 2.899×10^9 bytes
- 50 virtual workers: 2.976×10^9 bytes
- 100 virtual workers: 3.024×10^9 bytes

Total bytes uploaded to the global model on the improved model:

- 10 virtual workers: 4.512×10^9 bytes
- 50 virtual workers: 4.560×10^9 bytes
- 100 virtual workers: 4.800×10^9 bytes

The figures below show the cumulative sum of the bytes uploaded at each epoch. From the results, there is a substantial difference between the two models. The improved model achieved an average reduction of total bytes uploaded by 36% relative to the vanilla model.

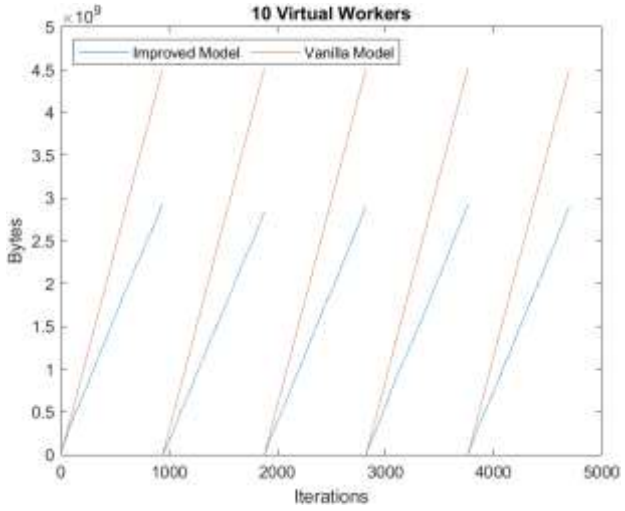

Figure 5.7

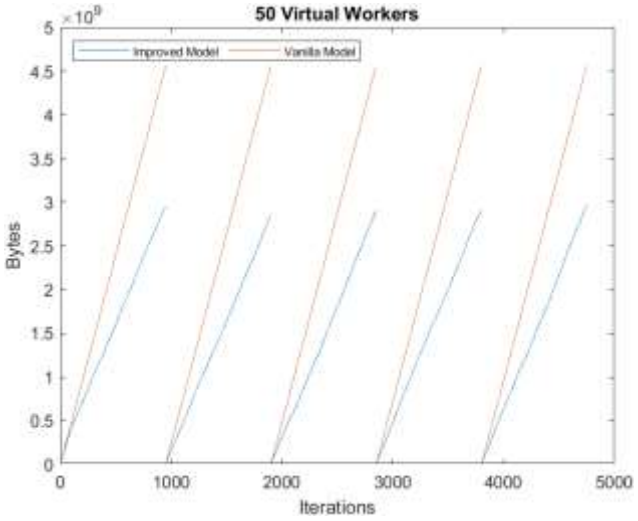

Figure 5.8

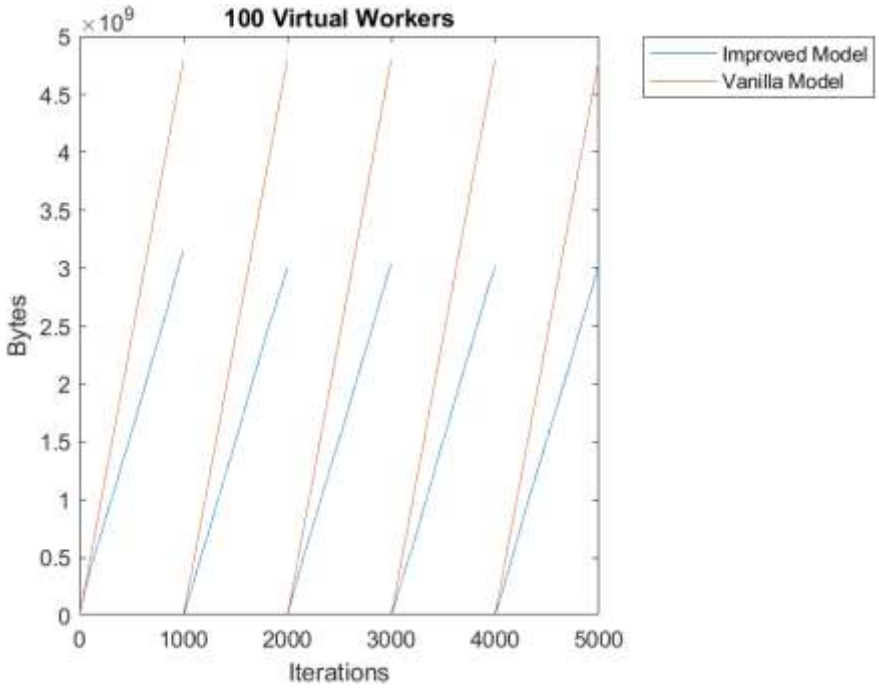

Figure 5.9

4.4 Final Analysis

Overall, it is no surprise that the vanilla model achieved better accuracy and loss results as it can aggregate over more iterations. Even though the improved model is not optimal for getting the best results, it significantly reduces the number of uploads sent and eases the burden on the network. In keeping this within a research scope, this improved model is ideal for federated learning systems that want to prioritise network efficiency and are not too concerned with the slight reduction in performance.

5 Professionalism and Responsibility

One of the fundamental reasons for federated learning being so popular is being to access local device's real-world data without infringing people's privacy and in the past, this would mean that people stayed clear from this area of work and so continued to learn models on abstract data such as classifying handwritten numbers. This new paradigm unlocks the potential of federated learning to use sensitive data in training models such as patient health records. This gives the ability to train models to spot symptoms quicker or correct medical professionals on a wrong diagnosis and this all can lead to a healthier population. However, to make it viable, the network needs to support this extra communication overhead.

My solution does address this problem but by its nature, it segregates a population of devices into a small sample. In my implementation, I have decided to use metrics that measure properties that cannot be able to identify the person so as not to discriminate against them. Furthermore, combining my solution with distributed dataset could lead the algorithm to select a few devices that are rich in data and will lead to the model to overly represent them, linking it back to the user. By relaxing the benchmarks in my algorithm to identify what's an acceptable model, we can include more of the population and make it more representative.

6 Conclusion and Future Research

6.1 Conclusion

Federated learning system is highly respected at the current time as it allows access to an extensive use of the people's private data whilst not infringing on their privacy. Allowing for models to be trained for complex tasks. Unfortunately, the current network is unable to support the increase in communication overhead. Therefore, to make this system viable, network resources need to be assigned in away to make full use of the network whilst achieving the best convergence of the model.

In this paper I have described how an algorithm that can be seamlessly integrated in the vanilla approach to solve this problem. The main aspect the algorithm has focused on is selecting certain local devices that contribute the most to a global model and rejecting unsatisfactory models. Simulating this approach showed that it does reduce the data sent thus reducing the communication overhead. However, this comes at the expense of a decrease in the quality of the model.

6.2 Future Research

I recommend that further research should be conducted in the selection of clients in an asynchronous federated learning system. This recreates the behaviours of edge devices in the real world, and this can contribute to whether which different types of devices should be chosen to make federated learning viable.

7 Bibliography

[1] J. Xu and H. Wang, "Client Selection and Bandwidth Allocation in Wireless Federated Learning Networks: A Long-Term Perspective," *IEEE Transactions on Wireless Communications*, vol. 20, no. 2. Institute of Electrical and Electronics Engineers Inc., pp. 1188–1200, Feb. 01, 2021. doi: 10.1109/TWC.2020.3031503.

[2] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A Survey on Distributed Machine Learning," *ACM Computing Surveys*, vol. 53, no. 2. Association for Computing Machinery, Jun. 01, 2020. doi: 10.1145/3377454.

[3] "THE POLICY CORNER."

[4] J. Park, S. Samarakoon, M. Bennis, and M. Debbah, "Wireless Network Intelligence at the Edge," Dec. 2018, [Online]. Available: http://arxiv.org/abs/1812.02858

[5] S. Hu, X. Chen, W. Ni, E. Hossain, and X. Wang, "Distributed machine learning for wireless communication networks: Techniques, architectures, and applications," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 3. Institute of Electrical and Electronics Engineers Inc., pp. 1458–1493, Jul. 01, 2021. doi: 10.1109/COMST.2021.3086014.

[6] E. Jeong, S. Oh, H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Communication-Efficient On-Device Machine Learning: Federated Distillation and Augmentation under Non-IID Private Data," Nov. 2018, [Online]. Available: http://arxiv.org/abs/1811.11479

[7] E. Horvitz and D. Mulligan, "Data, privacy, and the greater good," *Science (1979)*, vol. 349, no. 6245, pp. 253–255, Jul. 2015, doi: 10.1126/science.aac4520.

[8] B. Mahesh, "Machine Learning Algorithms-A Review," *International Journal of Science and Research*, 2018, doi: 10.21275/ART20203995.

[9] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553. Nature Publishing Group, pp. 436–444, May 27, 2015. doi: 10.1038/nature14539.

[10] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into Imaging*, vol. 9, no. 4. Springer Verlag, pp. 611–629, Aug. 01, 2018. doi: 10.1007/s13244-018-0639-9.

[11] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," Nov. 2015, [Online]. Available: http://arxiv.org/abs/1511.08458

[12] J. Gu *et al.*, "Recent Advances in Convolutional Neural Networks," Dec. 2015, [Online]. Available: http://arxiv.org/abs/1512.07108

[13] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, “Robust and Communication-Efficient Federated Learning from Non-IID Data,” Mar. 2019, [Online]. Available: http://arxiv.org/abs/1903.02891

[14] S. Niknam, H. S. Dhillon, and J. H. Reed, “Federated Learning for Wireless Communications: Motivation, Opportunities and Challenges,” 2020.

[15] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated Machine Learning: Concept and Applications,” Feb. 2019, [Online]. Available: http://arxiv.org/abs/1902.04885

[16] A. Brasoveanu, M. Moodie, and R. Agrawal, “Textual evidence for the perfunctoriness of independent medical reviews,” in *CEUR Workshop Proceedings*, 2020, vol. 2657, pp. 1–9. doi: 10.1145/nnnnnnn.nnnnnnn.

[17] J. Nguyen *et al.*, “Federated Learning with Buffered Asynchronous Aggregation,” Jun. 2021, [Online]. Available: http://arxiv.org/abs/2106.06639

[18] J. Goetz, K. Malik, D. Bui, S. Moon, H. Liu, and A. Kumar, “Active Federated Learning,” Sep. 2019, [Online]. Available: http://arxiv.org/abs/1909.12641

[19] T. Nishio and R. Yonetani, “Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge,” Apr. 2018, doi: 10.1109/ICC.2019.8761315.

[20] T. Ryffel *et al.*, “A generic framework for privacy preserving deep learning,” Nov. 2018, [Online]. Available: http://arxiv.org/abs/1811.04017

[21] T. Unterthiner, D. Keysers, S. Gelly, O. Bousquet, and I. Tolstikhin, “Predicting Neural Network Accuracy from Weights,” Feb. 2020, [Online]. Available: http://arxiv.org/abs/2002.11448