Parameter shift in Federated Learning for the IoT: Next word prediction on Raspberry Pi

Alexandr Goultiaev Tolstokorov, BAI

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Engineering (Electronic Engineering)

Supervisor: Meriel Huggard

April 2023

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Alexandr Goultiaev Tolstokorov

April 17, 2023

Permission to Lend and/or Copy

Alexandr Goultiaev Tolstokorov

April 17, 2023

Parameter shift in Federated Learning for the IoT: Next word prediction on Raspberry Pi

Alexandr Goultiaev Tolstokorov, Master of Engineering
University of Dublin, Trinity College, 2023

Supervisor: Meriel Huggard

Federated learning (FL) is a promising approach for training machine learning models in the Internet of Things (IoT) setting, where data is distributed across multiple devices. However, FL faces challenges such as heterogenous data distributions which induce a parameter shift in the trained models. This dissertation discusses this issue in depth and implements a number of strategies to mitigate parameter shift. The model implemented performs next word prediction on the Reddit dataset, and is implemented on Raspberry Pi's to simulate a real world FL deployment. Different parameters are experimented with to find a heuristically optimal baseline model which is compared to different configurations of Federated Batch Normalization (FedBN), Federated Proximal (FedProx) and federated optimizers. The results indicate that there is severe overfitting present in the FL system set-up and the parameter shift mitigations prove to be inconclusive in their effectiveness as their effect is overshadowed by the model's overfitting. Increasing the number of clients in the FL system to at least above 10 is necessary to get an acceptable performance of the model and witness the effect of parameter shift mitigations.

Acknowledgments

I would like to express my gratitude to Dr. Meriel Huggard, my supervisor, for her invaluable guidance and support throughout my dissertation. Dr. Huggard provided me with extensive feedback, insightful suggestions, and constructive criticism that significantly improved the quality of my work. Additionally, I would like extend my deepest appreciation to my family, friends and partner for their unwavering support, love, and encouragement throughout my academic journey.

Alexandr Goultiaev Tolstokorov

University of Dublin, Trinity College
April 2023

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet-of-Things (IoT) adoption and deployment rate has been rapidly increasing as of late, showing exponential growth potential facilitated by the integration of millions of interconnected IoT devices. IoT edge devices include different kinds of drones, robots, mobile devices and sensors, to name a few. These devices have limited communication and computation capabilities but also generate substantial amounts of data, and that volume is also increasing exponentially. To make use of this rich amount of data and enhance the end users experience, deep learning (DL) models are trained and deployed in these networks of devices. Conventional DL implementations are inseparable from the data that they are trained on and involve having the training data on the same node as the computation. This is an issue for IoT applications as the data can be subject to privacy concerns and communication between the edge device and server can be too costly for the edge device. These issues have now been addressed by the introduction of a distributed learning technique called Federated Learning (FL) [1] which eliminates the need to store the data and train the model centrally, instead delegating the model training to the edge devices where the server aggregates these edge device models to create the global model.

One standout example of applications to benefit from IoT networks of mobile devices is Natural Language Processing (NLP). NLP is a branch of Artificial Intelligence (AI) that deals with the interaction between computers and human languages. It focuses on the ability of computers to understand, interpret and generate human language both written and spoken. NLP plays an increasingly important role in our lives by making machines understand and communicate more naturally and intuitively with us. It is an essential component of many industries such as healthcare, finance, customer service, and marketing, to name a few. NLP has made huge progress with the proliferation of deep learning in recent years on tasks such as text processing, speech recognition and translation. One popular implementation of language models is next word prediction, where the model attempts to predict the next word in a sentence. A language model attempting next word prediction needs to be trained on text data from that device which is often private. This is an example where federated learning is useful in such applications as it allows the language model to train on the user device itself allowing the device's data to remain private.

1.1 Motivation

The successful commercial deployment of next word prediction for federated learning by Google[2] highlights the potential benefits of federated learning on mobile-devices. However, the next word prediction task in a federated learning setting brings up the issue of non-identically distributed data in each client. Because each user is its own independent entity, its data distribution is not guaranteed to be identical to ones from other users and therefore might introduce a parameter shift in its model. This means that the global model is aggregating parameter shifted models. This challenge can significantly affect the convergence and effectiveness of the model, which ultimately hinders its ability to provide accurate predictions. As such, addressing the problem of non-identically distributed data in a federated learning setting is critical to improving the performance and accuracy of models that rely on such data.

Therefore, this research aims to explore effective solutions to this issue in the task of next word prediction in a real world federated learning setting using Raspberry Pi as edge devices, with the ultimate goal of improving the performance of next word prediction models in federated learning settings. The Raspberry Pi is a small, low-cost and compact single board computer. In the context of the IoT, Raspberry Pi devices are often used as the brains of IoT projects. They can be used to collect data from sensors, process data and communicate with other devices or the cloud. Raspberry Pi's small size and low power consumption make it an ideal choice for IoT applications, where devices may need to be compact and energy efficient. As such they are the ideal candidate for implementing a next word prediction model in a federated learning setting and explore the effects of and solutions for non identically distributed data.

1.2 Goals and Objectives

The aim of this study was investigating the impact of non identically distributed data in next word prediction in a federated learning setting. Additionally, investigate different strategies from previous research in the field that tackle the issue of non-identically distributed data in FL and compare their effectiveness. A key objective was to conduct the study in a real-world scenario and avoid simulation, instead using Raspberry Pi's as the edge device in a FL system to obtain a realistic evaluation of the factors affecting the real-world FL systems.

The deep learning model had to be designed in accordance with the computing capabilities of the Raspberry Pi, the next word prediction task and the data on which it would train and predict on.

1.3 Dissertation Layout

Below is the outline of the five chapters of the dissertation:

Chapter 1: Introduction, motivation and goals of the research paper.

Chapter 2: A Literature Review, introducing and summarizing the research in the relevant fields for this study. A brief discussion of the challenges faced by FL and state-of-the-art solutions to those issues. Lastly, an overview of the FL frameworks available and challenges of implementation on resource-constrained devices such as the Raspberry Pi.

Chapter 3: Detailed overview of the design choices and implementation of the next word prediction model on Raspberry Pi's, the federated framework used and the data collection and prepossessing process.

Chapter 4: The results obtained and the establishment of a baseline benchmark. Comparison and evaluation of results of state-of-the-art solutions to the parameter shift challenge with the baseline.

Chapter 5: Concludes the dissertation with closing remarks and possible directions for future work.

1.4 Summary

This chapter discloses the motivations and goals for this research paper. A layout of the documents structure was provided summarizing the theme of each chapter. Lastly, an introduction section describes the setting for the project.

Chapter 2

Literature Review

This chapter introduces the concept of federated learning, how it works and its uses. It further discusses the most popular aggregation strategy used in FL termed Federated Averaging, then some of the issues and challenges that face federated learning and how they affect performance and adoption of the FL systems. From literature, some standout strategies to deal with these issues are explored, namely: FedBN, FedProx and adaptive federated optimizers. The chapter additionally introduces the task of next word prediction in the context of NLP. As such providing a background introduction of Recurrent Neural Networks (RNNs) and Long-Short-Term-Memory (LSTM) networks and their use in the next word prediction task. Furthermore, state-of-the-art implementations of FL in real devices are explored with the focus on mobile-devices and other resource-constrained devices such as the Raspberry Pi and the challenges of such implementations are highlighted. Lastly, this chapter provides a description of the most popular federated frameworks.

2.1 Federated Learning

Federated Learning is a distributed deep learning technique that enables multiple parties to collaboratively train a model without the need to share their data. FL has emerged as a promising approach to address privacy concerns and data security issues associated with centralized deep learning. The term federated learning in the context of machine learning was first coined by *McMahan et al.* [1] in 2016. Since then the field has grown massively and has focused on utilizing decentralized rich data spaces such as those available to IoT deployments. With successful commercial implementations such as the Gboard by *Hard et al.* [2] in 2018 for next word prediction in mobile-devices, healthcare and biomedical applications [3, 4, 5], finance [6] and autonomous driving [7], to name a few.
The general procedure of a FL system is the following:

1 Initialization: The parties agree on the machine learning model that they want to train collaboratively. The initial parameters are distributed to all parties.

2 Local training: Each party trains the model using its own data, thus avoiding sharing the data with other parties. The party then computes the model updates based on its local data and sends the updates to a central aggregator.

3 Aggregation: The central aggregator aggregates the model updates from all parties to compute a new set of global parameters. The aggregation can be done using a variety of strategies, with the conventional approach being Federated Averaging (FedAvg) described by *McMahan et al.* in the seminal field paper [1].

4 Model update: The new set of global model parameters is given out to all parties, and each party updates its local model using the new parameters. This process of local training, model update and aggregation is repeated in an iterative manner forever or until the model converges, depending on the application.

2.1.1 Federated Averaging

When proposing the concept of federated learning, *McMahan et al.* [1] two baseline algorithms for federated optimization are established: Federated Stochastic Gradient Descent (FedSGD) and Federated Averaging (FedAvg). These strategies outline the aggregation strategy and optimization method used in the model learning. Stochastic Gradient Descent (SGD) is a popular optimization algorithm used in machine learning and deep learning for training models. It is an iterative optimization algorithm that aims to find the optimal set of model parameters that minimizes a certain loss function. Simple gradient descent involves iteratively optimizing for the loss function by calculating the gradients of the model parameters at each data point. This becomes an issue with large datasets as the number of derivatives and calculations per step becomes unfeasible. In SGD, the derivatives are only computed on a small subset often called a mini-batch of the training data, rather than the entire dataset. This makes SGD an approximation of simple gradient descent that is more computationally efficient, especially on large datasets.

Applying SGD in the federated setting, a fraction of the clients C from the entire set K are selected each training round and compute the gradient of the loss over all the data held by these clients. In this approach C is a hyperparameter that controls the global batch size, with $C = 1$ being non-stochastic gradient descent. The model parameters from these updates is then aggregated at a server by naive averaging, this approach is termed FedSGD. Federated Averaging is based on FedSGD and allows for clients to perform multiple iterations of the local model update before the server aggregation step.

In FedAvg the amount of computation is decided by three hyperparameters: C the fraction of clients that participate in training each round, E the number of training passes that each client does on its local data (epochs) and B the local mini-batch size used for the client updates.

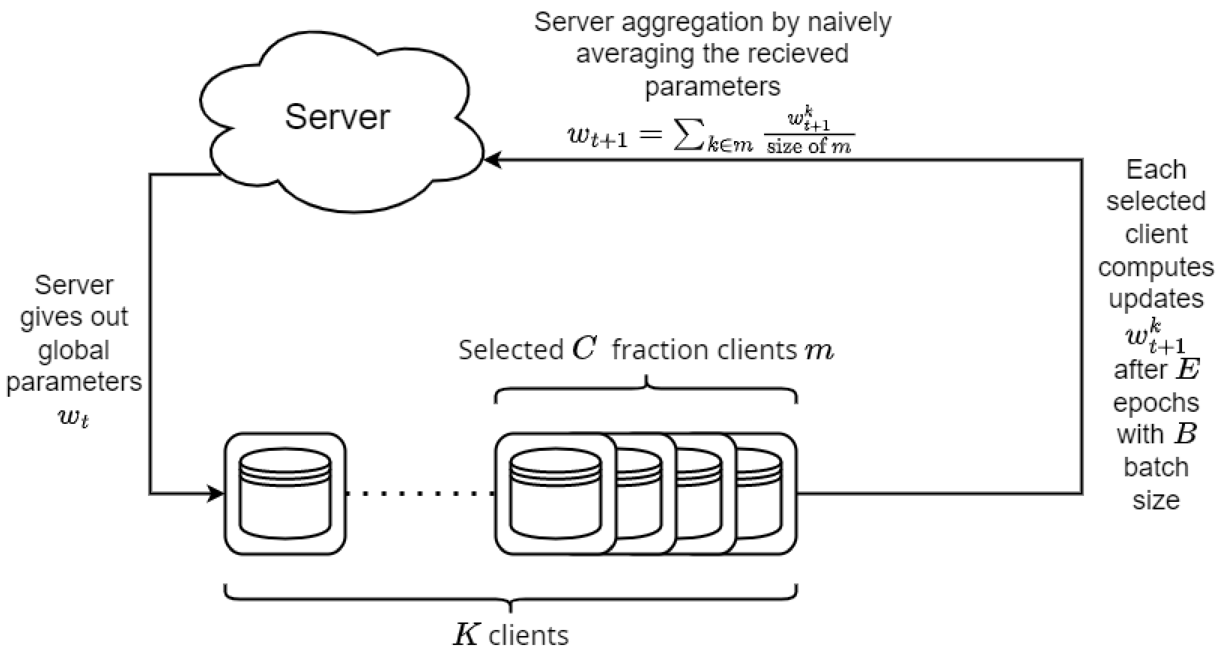

Server aggregation by naively averaging the recieved parameters
$w_{t+1} = \sum_{k \in m} \frac{w_{t+1}^k}{\text{size of } m}$
Server
Server gives out global parameters w_t
Each selected client computes updates w_{t+1}^k after E epochs with B batch size
Selected C fraction clients m
K clients

Figure 2.1: This diagram shows a Federated Averaging training round. The server first gives out the global parameters w_t to a subset C of the K clients. In the case that this is the first training round $w_t = w_0$ where w_0 is the random initialization of model parameters. The selected clients proceed to compute local updates over E epochs using B batch size. These clients then send their updated parameters w_{t+1}^k to the server. Finally, the server averages the recieved model updates and produces a new global parameter set w_{t+1} which is then sent down to the next subset of clients to repeat the process.

2.1.2 Issues and challenges with Federated Averaging

Federated Averaging turned out to have issues in practical usage with the most common problems being: communication bottlenecks at larger scales, privacy weaknesses and computation requirements on the edge devices. A number of papers reviewing the field and scientific surveys (*Li et al.*[8](2020), *Lim et al.*[9](2020) and *Kairouz et al.*[10](2021)) have identified some of the main areas of research in the field to overcome the limitations of the naive averaging aggregation approach, them being: communication optimization, preserving privacy, ensuring fairness, robustness to failures, ability to deal with non-iid data and computation requirement at edge devices.

Communication optimization

Communication cost is often a bottleneck in FL when scaling up the network of distributed devices collaborating to learn a model, with some devices potentially also having unreliable or limited communication. Strategies that are often used in algorithms optimizing communication in literature are periodic averaging where devices compute a number of updates before sending their parameters for aggregation, effectively reducing the number of communications, and compression (*Reisizadeh et al.*[11](2020), *Haddadpour et al.*[12](2021)). Additionally compression can be implemented together with offloading some of the computation to the server (*Rothchild et al.*[13](2020)). Alternatively, some approaches trade a higher computation load at the device for lower communication load, such as FedDyn by *Acar et al.*[14](2021) which opts to use a dynamically set regularizer term to each device to, in the limit, align its model to the global one.

Ensuring fairness

In FL, computational power of all devices part of it is exploited which allows us to get a richer model with a larger set of training data. However, this means that we develop a common output for all the devices and thus, it does not necessarily adapt the model to each user. In the case of heterogeneous data distributions for the devices this problem is exacerbated. To circumvent this and obtain a more uniform accuracy distribution across the devices an effective solution is an algorithm to set weights to worse performing clients so that their updates have more influence and therefore shift the accuracy distribution towards a more even distribution, this is done in q-FedAvg by *Li et al.*[15](2019). Some model-agnostic approaches operate by first finding an initial shared model where each device can then perform a few steps of gradient descent to adapt its own parameter set to (*Fallah et al.*[16](2020)), while others only aggregate common data to the server model and every device then trains in a backbone net that is personalized for itself such as in FedGB by *Yang et al.*[17](2020).

Robustness to failure

Due to the distributed nature of the computation in FL systems and the heterogeneous nature of the target devices both in communication and in computation capability, some devices may fail either partially or fully to contribute to the server parameter aggregation. These devices that fail are called stragglers. To deal with stragglers literature either changes the aggregation strategy or attempts to identify the stragglers and deal with them directly. An example of a solution that changes the overall strategy is detailed by *Chen et al.*[18](2020) in which the conventional synchronous aggregation of every device´s parameters each round is changed to accommodate stragglers updating asynchronously. Otherwise, stragglers can be identified and dealt with such as using Helios an FL system introduced by *Xu et al.*[19](2021) which works by identifying the stragglers in a system and assign them a simplified version of the global model to train. Devices that are malicious actors in a FL system are called adversaries, research is also conducted in how to defend against these adversaries. As an example *Park et al.*[20](2021) introduce Sageflow which intends to tackle stragglers by performing model grouping and weighting the devices at aggregation according to their staleness (update arrival delay) and tackle adversaries by employing entropy based filtering and loss-weighted averaging of parameters at every grouping stage.

Computation requirement at edge devices

FL delegates computation from a central server node to the edge devices and this is its main strength. However, this also means that each device has to run and train a copy of the full server model which might limit the available choices for neural networks to conform with the computation capability of the devices, or alternatively limit the acceptable devices in a FL system. To deal with this constraint two main solutions are emerging: having devices train on smaller sub-models or adaptive pruning of the global model. Having devices train on smaller sub-models of the global full model has been implemented in different ways: using federated dropout to train the devices on sub-models of the global model as described by *Caldas et al.*[21](2018), using a distillation of knowledge from the smaller device models to the global model such as FedGKT introduced by *He et al.*[22](2020) or selectively using a simplified model for training devices that are more resource limited such as in *Rapp et al.*[23](2020) where each device has a neural network with a computationally intensive and simpler paths. On the other hand, there is a pruning approach where the global model is dynamically pruned to be the most concise version of itself while still serving the devices with high accuracy such as in PruneFL by *Jiang et al.*[24](2020) where the model is pruned so that the computational load on the devices is reduced. Recently some papers bridging split learning (SL) and FL have also demonstrated advantages when mitigating the computational load on participant devices such as the paper by *Thapa et al.*[25](2022) which uses SL methods together with devices training on sub-models to not only alleviate the computational load but also improve privacy.

Ability to deal with non-iid data

Finally, the data available to each device can also be of heterogeneous nature due to either differences of data acquisition of each device or a multitude of other factors such as location and environment, sensor type, etc... Therefore, naively aggregating the parameters of devices trained on non-independently identically distributed data (non-iid) would hurt convergence of the model. A non-iid data distribuition introduces what is called a parameter shift when a device is trained on it, to alleviate the parameter shift multiple solutions have been proposed, but the main goal is minimizing the impact of devices with parameter shift in the network. Some of the state-of-the-art solutions to this issue that I will be implementing in this dissertation are FedBN by *Li et al.*[26](2021), FedProx by *Li et al.*[27](2020) and Adaptive federated optimizers by *Reddi et al.*[28](2020).

2.1.3 FedProx

Federated Proximal (FedProx) is a federated learning strategy that addresses the challenge of non-iid data distribution in a FL system. FedProx introduces a regularization term to the standard federated learning optimization objective, in order to account for the parameter shift across clients. The regularization term, termed the proximal term, encourages the local model at each client to stay close to the global model by acting as a regularization penalty that discourages the local models from deviating too far from the global model. The FedProx optimization objective h_k of a client k can be formulated as follows:

$$\min_{w} h_k(w_t; w_{t+1}) = l_k(w; D_P) + \frac{\mu}{2}||w_t - w_{t+1}^k||^2 \tag{2.1}$$

2.1: FedProx optimization objective

Where $l_k(w; D_P)$ is the loss function of that client, measuring the performance of the local model with parameters w on the local data D_P. The proximal term μ is multiplied with the Euclidean distance between the local model parameters after computation w_{t+1}^k and the global model parameters w_t.

FedProx has been shown to improve the performance and robustness of federated learning algorithms, especially in scenarios where data distribution across parties is highly heterogeneous.

2.1.4 FedBN

Federated Batch Normalization (FedBN) is a federated learning strategy that adopts batch normalization, a popular technique used in deep learning for improving training stability and accelerating convergence. Batch normalization is a technique first proposed by *Ioffe et al.*[29] in 2015, the technique normalizes the mean and standard deviation of the inputs within a mini-batch of samples during training. This helps to mitigate the internal covariate shift, which can occur when the distribution of inputs to a neural network layer changes during training, which can slow down the training process.

FedBN extends the concept of batch normalization to the federated learning setting by allowing clients of a FL system to compute and update the batch normalization statistics (i.e. mean and standard deviation) in a distributed manner, while preserving data privacy. The core idea of FedBN is to allow each client to update its local batch normalization statistics using its own local data during training, and then use a federated aggregation strategy such as FedAvg to combine the local model parameters excluding the batch normalization layers. FedBN helps mitigate the challenges of non-iid data distribution in FL systems.

2.1.5 Adaptive federated optimizers

Adaptive Federated Optimization is a strategy proposed that dynamically adapts the optimizer function during the training process. In conventional federated learning approaches, the hyperparameters such as the learning rate remain fixed throughout the training process across all clients and both the clients and the server use SGD optimizers. In [28] a strategy termed FedOpt is proposed. FedOpt rewrites the FedAvg update as follows:

$$w_{t+1} = \frac{1}{|m|}\sum_{k \in m} w_{t+1}^k = w_t - \frac{1}{|m|}\sum_{k \in m}(w_t - w_{t+1}^k) \tag{2.2}$$

2.2: FedOpt update

Where w_t are the global model parameters sent out that training round, w_{t+1} is the aggregated new global model parameters and w_{t+1}^k are the model parameters of client k after the training round. The selected subset of clients in the training round is m with $|m|$ being the amount of clients in that subset. Let $\Delta_t^k = w_t^k - w_t$ and $\Delta_t = \frac{1}{|m|}\sum_{k \in m}\Delta_t^k$. Then the server update in FedAvg is equivalent to applying SGD to the "pseudo-gradient" $-\Delta_t$ with learning rate $\eta = 1$.

Therefore in FedOpt the server can use an adaptive optimization method such as ADAGRAD[30], YOGI[31] or ADAM[32] to optimize the "pseudogradient" laid out in Equation 2.2, while the client is using SGD for its loss function to update its model parameters.

2.2 Natural Language processing

Natural Language Processing is a dynamic field that has evolved significantly over the years, with notable events shaping its development. The origins of NLP can be traced back to the 1950s and 1960s, with foundational works such as "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence" by *McCarthy et al.*[33] (1955) and "Three Models for the Description of Language" by *Noam Chomsky*[34] (1956). These early works laid the groundwork for the development of computational models for language processing. The next stage were rule-based NLP systems in the 1970s and 1980s, rule-based NLP systems gained popularity, with notable works such as "Natural Language Understanding" by *James Allen*[35] in 1988. Rule-based systems used handcrafted grammatical rules to analyze and generate natural language, which were limited by their rigid structure and lack of scalability. The 1990s saw the rise of statistical NLP and machine learning approaches which revolutionized the field. Notable works include "Foundations of Statistical Nature Language Processing" by *Manning et al.*[36] in 1999 and "Maximum Entropy Models for Natural Language Ambiguity Resolution" by *Adwait Ratnaparkhi* in 1998. These approaches made use of large data sets and statistical models to learn patterns from text leading to significant advances.

Finally, in the past decade, deep learning has emerged as a dominant paradigm in NLP, with breakthroughs in areas such as word embeddings, recurrent neural networks and transformer models. Notable works include "Word2Vec: Efficient Estimation of Word Representations in Vector Space" by *Mikolov et al.*[37] in 2013, "Sequence to Sequence Learning with Neural Networks" by *Sutskever et al.*[38] in 2014, and "Attention is All You Need" by *Vaswani et al.*[39] in 2017. These works have paved the way for modern NLP techniques, such as neural machine translation, text classification and sentiment analysis. Recent advances in transfer learning and pre-trained models have further improved the state-of-the-art in NLP such as "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" by *Devlin et al.*[40](2019) and "ULMFiT: universal Language Model Fine-tuning for Text Classification" by *Howard et al.*[41](2018). These approaches leverage large pre-trained models on vast amounts of data to achieve impressive performance on a wide range of NLP tasks with limited amounts of task-specific data.

2.2.1 Next Word Prediction

As a subset task in NLP, next word prediction has also benefited from the popularization of deep learning leading to significant advancements and new state-of-the-art approaches. Notably, Recurrent Neural Networks have proven to be useful for next word prediction.

RNNs differ from traditional Feed Forward Neural Networks (FFNN) in that they allow feedback connections between neurons and layers allowing information to propagate both ways. This allows RNNs to have a "memory" as the state of the neuron not only depends on the current value but on the past states of the network [42]. This allows RNNs to learn temporal tasks as the nonlinear transformations of the input signal that happen inside the network capture useful contextual information. Thus, RNNs are able to learn sequential text data and perform classification and prediction tasks.

2.2.2 Long Short Term Memory Networks

Long Short Term Memory (LSTM) networks are a type of RNN architecture that are designed to overcome the limitations of traditional RNNs, such as the vanishing gradient and enable better handling of long-range dependencies in sequential data. LSTMs were introduced by *Hochreiter et al.*[43] (1997). LSTMs are useful in NLP and next word prediction as they can learn patterns from input sequences of varying lengths and can remember information over long sequences thus being well-suited to model temporal dependencies. LSTMs have been shown to achieve state-of-the-art performance in next word prediction such as "LSTM recurrent networks learn simple context-free and context-sensitive languages" by *Gers et al.*[44] (2002) which was one of the earliest applications of NLP showing that their LSTM-based language model outperformed traditional n-gram language models on several benchmark dataset. More recently "Language Models are Unsupervised Multitask Learners" by *Radford et al.*[45] (2019) proposed the GPT-2 (Generative Pre-trained Transformer 2) model, which is a large-scale language model based on the transformer architecture with LSTM-based decoders. The GPT-2 model achieved state-of-the-art performance on several NLP benchmarks, including next word prediction.

2.3 Federated Learning implementations on real devices

Federated learning has had a number of notable industry and commercial implementations on real devices. The most famous of which is the Gboard[2] which uses federated learning on mobile devices for next word prediction on typing keyboard. The paper evaluates a LSTM network and finds that it is well suited for the task performing better than older techniques such as finite state machines and simple RNNs. They find that using the LSTM network also reduces the number of parameters needed therefore reducing the computational complexity at edge devices without having an adverse effect on accuracy.

2.3.1 Limitations and challenges of using Raspberry Pi

Raspberry Pi is a series of small single board computer developed by the Raspberry Pi Foundation. It is designed to be affordable, compact and versatile, making it a popular choice for a wide range of applications, including hobbyist projects, prototyping and IoT projects. They are also widely used in educational settings to teach programming, electronics, and computer science concepts to students. The components of a Raspberry Pi are an ARM processor, program memory (RAM) , storage from an external mini-SD card, input/output ports, and various connectivity options, such as USB and USB-C, Ethernet, mini-HDMI, and Wi-Fi. They run on Linux-based operating systems, and can be programmed using various programming languages, making them accessible to a wide range of users with different levels of technical expertise.

For these reasons, the Raspberry Pi is a common staple of research papers and trial implementations of FL deep learning on resource-constrained devices. However, there are some challenges associated with implementations on the Raspberry Pi, namely:

Limited computational power: Raspberry Pi boards are low-power and low-performance devices compared to high-end GPUs or other dedicated deep learning hardware. The limited computational power of Raspberry Pi's can limit the complexity and size of models that can be effectively trained and deployed on the devices, leading to potential performance limitations.

Limited storage capacity: Raspberry Pi boards may have limited storage capacity, which can limit the amount of data that can be stored locally on the devices for training. This can impact the ability to store and process large datasets required for FL applications, leading to potential data storage limitations[10].

Power limitations: Raspberry Pi boards are powered by a USB connection or a power adapter, which may not provide sufficient power for running computationally-intensive FL models for prolonged periods of time. This can lead to potential power-related issues, such as reduced performance or device shutdowns.

Limited I/O capabilities: Raspberry Pi boards have limited I/O capabilities compared to higher-end devices, which can affect the ability to connect and interface with external sensors, devices or peripherals that may be required for certain FL applications. This can lead to potential limitations in data collection or communication between devices[46].

Heat dissipation: Due to their small form factor, Raspberry Pi boards may not have adequate heat dissipation, leading to potential overheating issues when running computationally-intensive FL models for prolonged periods of time. This can impact the devices performance and reliability.

2.4 Federated Learning frameworks

The growth of the field of FL has led to the development of several open-source frameworks for implementing federated learning. Some of the most notable being:

Tensorflow Federated(TFF)[47]: TFF is an open-source federated learning framework developed by Google. TFF offers high-level abstractions for defining federated learning algorithms and supports both server-side and client-side aggregations. TFF has a large community of developers and users, which ensures regular updates and improvements.

PySyft[48]: PySyft is an open-source federated learning framework developed by OpenMined. PySyft is built upon PyTorch, a popular deep learning framework, and extends it with federated learning capabilities. PySyft offers advanced features like differential privacy and secure multi-party computation, making it suitable for privacy-sensitive use cases.

FedML[49]: FedML is an open-source federated learning framework developed by researches from various institutions, including Carnegie Mellon University and Stanford University. FedML includes features like data prerpocessing, model aggregation and evaluation, making it a comprehensive framework for federated learning experiments.

Flower[50]: Flower is also an open-source federated learning framework developed by Adap, a company specializing in federated learning solutions. It provides a simple and lightweight platform for federated learning, with support for various machine learning frameworks, including TensorFlow and PyTorch. Flower offers high-level abstractions for defining federated learning strategies.

2.5 Summary

This chapter provides a literature review, organizing and summarizing the existing research and notable achievements in the fields of NLP and FL. The chapter provided a description of FedAvg the baseline agreggation algorithm for FL, additionally discussing the challenges it faces and notable research to help alleviate those issues. FedBN, FedProx and adaptive federated optimizers were introduced and discussed as FL strategies that aim to combat the issue of non-iid data distributions between clients. An introduction and history of landmark research in the field of NLP and the task of next word prediction was discussed, culminating in a review of a successful commercial implementation of FL on mobile devices to perform next word prediction. Raspberry Pi was introduced as a resource-constrained device, together with a detailed description of the challenges of implementing FL on Raspberry Pi's. Finally, the most popular federated frameworks were listed and introduced. The background research detailed in this chapter will allow the reader to better understand and appreciate the design choices and evaluation of the later chapters.

Chapter 3

Design and Implementation

In this chapter the design of my implementation of next word prediction FL on a Raspberry Pi is introduced. This chapter uses the background information laid out in the literature review to discuss the implementation design choices and any considerations regarding them. The chapter goes over in detail about the LSTM model architecture used, additionally introducing the concept of Layer Normalization and Recurrent Dropout in LSTM and how it will be implemented. An overview of the used optimization algorithms is given. The choice of federated framework is outlined and the inner workings of the flower framework are highlighted. Additionally, the training and testing data collection and preprocessing steps are explained. Lastly, the setup of the Raspberry Pi's for the FL implementation is described.

3.1 Model architecture

As introduced in the literature review chapter, LSTMs are a type of recurrent neural network that can model sequential data by capturing long-range dependencies. They use a specialized architecture with cells, hidden states, and gates to effectively store and update information over long sequences of data, making them useful for tasks that require modelling of sequential data where the order matters such as text. A LSTM cell is made up from three separate gates, that dictate the information to be stored and discarded in the cell state. These are the input, output and forget gates. Illustrated in Figure 3.1 is a diagram of the LSTM cell which will be used to describe the different gates and their functions.

The gates perform the following computations:

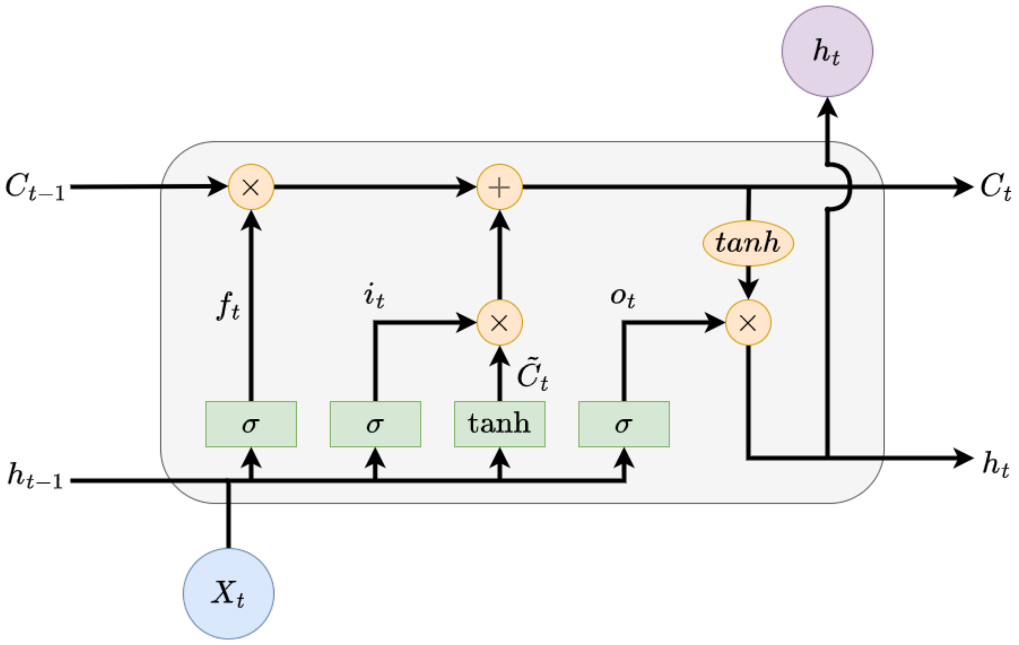


Figure 3.1: A diagram of a LSTM cell. Where green rectangular elements are neural network layers, in which σ denotes sigmoid activation and *tahn* denotes a tahn activation. The circular or elliptical orange elements denote point-wise operations. The output of the three gates are denoted as f_t for the forget gate, i_t for the input gate and o_t for the output gate. Finally the data input at time t to the cell is X_t, while C_t is the cell state at time t and h_t is the hidden cell state at time t.

Forget gate: The forget gate decides what information to throw away from the cell state. It performs this by a sigmoid layer using the current input to the cell X_t and the hidden state h_{t-1}, outputting a number between 0 and 1 for each number in the cell state C_{t-1}. The output of the forget gate is therefore: $f_t = (W_f \cdot [h_{t-1}, X_t] + b_f)$, where W_f are the forget gate weights and b_f is a bias term.

Input gate: The input gate decides what new information we store in the cell state. First, a sigmoid layer decides which values to update: $i_t = \sigma(W_i \cdot [h_{t-1}, X_t] + b_i)$ where W_i are the input gate weights and b_i is a bias term. Then the tahn layer creates a vector of new candidate values $\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, X_t] + b_C)$ to be added to the cell state, where W_C are candidate value weights and b_C is another bias term. After the input gate we have effectively updated the old cell state C_{t-1} into the new cell state $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$.

Output gate: Latstly, the output gate decides what we will output from this cell. The output will be a filtered version of our cell state. First, a sigmoid layer will decide which parts of the cell state will be in the output: $o_t = (W_o \cdot [h_{t-1}, X_t] + b_o)$ where similarly to before W_o are the weights of the output gate and b_o is a bias term. Finally, the updated cell state is passed through a tanh function to push the values in the range of -1 to 1, and multiply by the output o_t to create the updated hidden cell state: $h_t = o_t * tanh(C_t)$.

3.1.1 Layer Normalization in LSTM

Batch normalization as introduced in the literature review is a popular technique for speeding up training of deep learning models. However, the effect of batch normalization is dependent on the mini-batch size and it is not obvious how to apply it to a recurrent neural networks such as the LSTM without having a separate batch normalization coefficients for each time step. A paper by *Ba et al.*[51](2016) proposed a transpose of traditional batch normalization into layer normalization which is applicable to RNNs.

A conventional feed-forward neural network is a non-linear mapping from an input pattern $\mathbf{x}$ into an output vector $\mathbf{y}$. Consider the n^{th} hidden layer in a deep feed-forward neural network, let a^n be the vector representation of the summed inputs to the neurons of that layer.

$$a_i^n = w_i^n \top h^n \qquad h_i^{n+1} = f(a_i^n + b_i^n) \tag{3.1}$$

3.1: Where the summed input of a node i, a_i^n is a linear projection of the weight of that node w_i^n and the bottom-up inputs h^n. Where the bottom-up inputs h^n is given by a element-wise non-linear function $f(\cdot)$ and b_i^n is a scalar bias parameter.

These parameters are learnt using gradient based optimization algorithms using back-propagation. Batch normalization used the mean input of a node μ_i^n and the standard deviation of that node σ_i^n to normalize the summed input in the n^{th} layer.

$$\tilde{a}_i^n = \frac{g_i^n}{\sigma_i^n}(a_i^n - \mu_i^n) \tag{3.2}$$

3.2: Where $\tilde{a}_i^n$ is the normalized summed inputs to the i^{th} hidden node in the n^{th} layer and g_i is a gain parameter scaling the normalized activation before the non-linear activation function. Let the mean be $\mu_i^n = \mathbf{E}_{\mathbf{x} \sim P(\mathbf{x})}(a_i^n)$, and let the standard deviation be $\sigma_i^n = \mathbf{E}_{\mathbf{x} \sim P(\mathbf{x})}(a_i^n - \mu_i^n)^2$.

Typically μ and σ are estimated using a mini-batch and not the entire training dataset which is why it is hard to apply to RNNs. Layer Normalization fixes the mean and varience of the summed inputs within each layer, computing the normalization statistics over all the hidden nodes in the same layer as follows:

$$\mu^n = \frac{1}{H}\sum_{i=1}^{H} a_i^n \qquad \sigma_i^n = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^n - \mu_i^n)^2} \tag{3.3}$$

3.3: Where H is the total number of nodes in that layer.

Therefore if we use the LSTM cell illustrated in Figure 3.1 the layer normalized LSTM cell would be:

$$h^t = f[\frac{g}{\sigma^t} \cdot (a^t - \mu^t) + b] \qquad \mu^t = \frac{1}{H}\sum_{i=1}^{H} a_i^t \qquad \sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^t - \mu^t)^2} \tag{3.4}$$

3.4: Where we let $a^t = W_{hh}h^{t-1} + W_{xh}\mathbf{x}^t = \begin{pmatrix} f_t \\ i_t \\ o_t \\ \tilde{C}_t \end{pmatrix}$.

3.1.2 Recurrent dropout in LSTM

Neural network models often suffer from overfitting, especially in the cases where the amount of training data is small compared to the number of model parameters. This has lead to a lot of research into improving the generalization ability of deep models. One of the more popular methods to avoid overfitting is dropout regularization proposed by *Hinton et al.*[52](2012). However, dropout in input-to-hidden and hidden-to-output layers is suited for feed-forward networks and not so much for RNNs. A number of papers have proposed dropout regularization for RNNs (*Moon et al.*[53](2015), *Gal et al.*[54](2016)). However, these dropout regularizations introduce long-term memory corruption in LSTM networks. Finally, a paper by *Semeniuta et al.*[55](2016) introduced a dropout regularization with no long-term memory loss focused for LSTM networks which will be used in this thesis. Dropout in conventional RNN cells is applied as follows:

$$h_t = f(W_h[x_t, d(h_{t-1})] + b_h) \tag{3.5}$$

3.5: Where d is the dropout function defined as follows: $d(\mathbf{x}) = mask * \mathbf{x}$ if in the train phase and $d(\mathbf{x}) = (1-p)\mathbf{x}$ otherwise. p is the dropout rate and $mask$ is a vector sampled form the Bernoulli distribution with success probability $1-p$.

The approach suggested from [55] performs the dropout to the cell update candidate values $\tilde{C}_t$ of a LSTM cell. Applying this recurrent dropout together with the cell state update defined earlier in Figure 3.1:

$$C_t = f_t * C_{t-1} + i_t * d(\tilde{C}_t) \tag{3.6}$$

3.6: Using the same notation defined earlier when describing the LSTM gates and cell state update process.

3.2 Optimization algorithms.

Deep learning optimization algorithms are used to optimize the training process of deep networks by updating the model's parameters iteratively during training. Some of the popular optimization algorithms that I will use in my experiments are Adagrad[30], Adam[32] and Yogi[31]. Consider the FedOpt update from Equation 2.2: $w_{t+1} = w_t - \Delta_t$, in which $\Delta_t = \frac{1}{|m|}\sum_{k \in m} \Delta_t^k$ where $\Delta_t^k = w_t^k - w_t$. Using this update we can formulate the optimization algorithms as follows:

$$v_t = v_{t-1} + \Delta_t^2 (\mathbf{FEDADAGRAD}) \tag{3.7}$$
$$v_t = v_{t-1} - (1-\beta_2)\Delta_t^2 sign(v_{t-1} - \Delta_t^2)(\mathbf{FEDYOGI}) \tag{3.8}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\Delta_t^2 (\mathbf{FEDADAM}) \tag{3.9}$$

Let the updated new parameters be:

$$w_{t+1} = w_t + \eta \frac{z_t}{\sqrt{v_t} + \tau} \tag{3.10}$$

3.10: Where $z_t = \beta_1 z_{t-1} + (1-\beta_1)\Delta_t$, η is the learning rate, β_1 and β_2 are exponential decay parameters and τ is a smoothness term called adaptivity parameter.

3.3 Federated framework selection

The federated frameworks that were considered for this project were listed in the literature review in Section 2.4. The first framework to be discarded was Tensorflow federated (TFF). While I am most comfortable with Tensorflow out of all the deep learning libraries, TFF is only available for simulation and does not have the ability to be implemented in real devices which is the goal of this project. The first framework to be considered and experimented with was PySyft. One of the most popular web blogs detailing an implementation of FL on Raspberry Pi's uses PySyft[56]. However, after trying to implement it following the blog instructions I ran into errors which were due to a general update of the framework in which multiple of the functionalities were changed. Due to the recency of the update the documentation was not up to date, which ultimately made me reconsider using PySyft. Lastly, the other two frameworks considered that allow for real device implementation were FedML and Flower. FedML was considered but ultimately discarded in favour of Flower for its simplicity. While Flower's simplicity comes at a cost of the aggregation strategies that I can implement, ultimately the strategies that were chosen to implement were compatible with Flower and therefore its simplicity would allow for more time spent on exploring the effect of said strategies instead of spending more time getting accustomed to the more powerful but complex FedML framework.

3.3.1 Flower framework

One of the main strengths of the Flower framework is its model and device agnosticism[50], which allows for implementations with most of the popular deep learning frameworks and any edge device without having to modify the deep learning model. This allows for deployment of FL systems for heterogeneous client pools and easier scalability of existing FL systems as integration of a new device would require minimal adjustment.

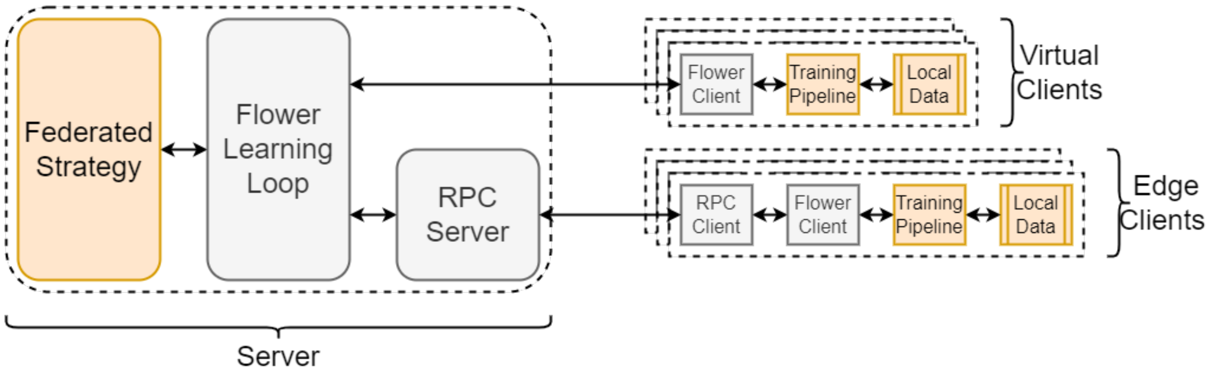


Figure 3.2: A diagram of the Flower framework architecture. The orange colored elements are user code segments while grey colored elements are the framework code. Shown in the diagram is the pipeline of an implementation using virtual and edge devices.

The Flower Learning Loop depicted in Figure 3.2 is responsible for orchestrating the federated learning process by executing global computations such as parameter configuration and aggregation using the defined Federated Strategy and local computations by passing training instructions to the clients. Any embedded device can be communicated with through a RPC server-client connection. As an example the paper by *Mathur et al.*[57](2021) explores the Flower framework for deployment on heterogeneous devices using Raspberry Pi's and Nvidia Jetson embedded devices. The Flower Client can then use any Python training pipeline for its model such as Tensorflow, PyTorch, Sklearn or even raw numpy, to name a few. This flexible and modular design greatly reduces the complexity of implementing FL, with basic implementations only requiring a few lines of code to federate an existing deep learning model.

3.4 Data collection and preprocessing

The objective of this paper is to implement a federated next word prediction model, simulating the next word prediction FL model present in mobile devices such as the Gboard. In their implementation [2], keyboard chat logs of mobile phone users were used after making sure to remove sensitive information. This approach also came about with a natural data split as each user had its own data on its device. To mimic this approach a similar dataset comprised of user written text was needed, importantly it should be partitioned in some sort of way resembling text written by separate entities so that the effect of non-iid data distribution can be evaluated and addressed with the planned strategies.

A dataset of Reddit[58] comments was ultimately chosen to be used in the implementation. The dataset is provided by LEAF[59] a modular benchmarking framework for FL implementations which preprocessed the dataset from [60] which include posts and comments from December 2017. Reddit is a forum website that hosts discussions divided across subreddits of different topics or events. This dataset contains a total of 56,587,343 comments from 1,660,820 users, however for this work only a subset of the dataset was used. The main reason for the choice of this dataset is that it closely resembles regular typing patterns that mobile users would generate and it is naturally partitioned into different users. Each user has a varying amount of comments and therefore training data and the data distribution of each user is also varying as each user has their own typing pattern which will let us explore the effect of non-iid data distributions in the FL implementation. For the implementation the dataset was preprocessed to be fed into the deep learning model. The dataset JSON file illustrated in Table 3.1 contains many unnecessary data fields which were removed, leaving only the author and body fields as we are only interested in the body of the comments for the training and testing of our LSTM model and the user ID to make a partition of data for each Raspberry Pi in our FL system. The steps taken to further preprocess the data are the following:

Removed any non-ASCII symbols: This was done to avoid problems with tokenization of the text later on.

Lowercase the text: The text of all comments was converted to lowercase, this avoids unnecessary complexity and issues of having to predict the uppercase or lowercase

Field	Description
id	The comment's identifier, e.g., "dbumnq8" (String).
author	The account name of the poster, e.g., "example username" (String).
link-id	Identifier of the submission that this comment is in, e.g., "t3 5l954r" (String).
parent-id	Identifier of the parent of this comment, might be the identifier of the submission if it is top-level comment or the identifier of another comment, e.g., "t1 dbu5bpp" (String).
created-utc	UNIX timestamp that refers to the time of the submission's creation, e.g., 1483228803 (Integer).
subreddit	Name of the subreddit that the comment is posted. Note that it excludes the prefix /r/. E.g., 'AskReddit' (String).
subreddit-id	The identifier of the subreddit where the comment is posted, e.g., "t5 2qh1i" (String).
body	The comment's text, e.g., "This is an example comment" (String).
score	The score of the comment. The score is the number of upvotes minus the number of downvotes. Note that Reddit fuzzes the real score to prevent spam bots. E.g., 5 (Integer).
distinguished	Flag to determine whether the comment is distinguished by the moderators. "null" means not distinguished (String).
edited	Flag indicating if the comment has been edited. Either the UNIX timestamp that the comment was edited at, or "false".
stickied	Flag indicating whether the submission is set as sticky in the subreddit, e.g., false (Boolean).
retrieved-on	UNIX timestamp that refers to the time that we crawled the comment, e.g., 1483228803 (Integer).
gilded	The number of times this comment received Reddit gold, e.g., 0 (Integer).
controversiality	Number that indicates whether the comment is controversial, e.g., 0 (Integer).
author-flair-css-class	The CSS class of the author's flair. This field is specific to subreddit (String).
author-flair-text	The text of the author's flair. This field is specific to subreddit (String).

Table 3.1: Dataset JSON object structure.
Table from [60] that lists the data fields in the JSON object and describes what each data field is.

variant of the same word. The task is next word prediction so the format of the word is irrelevant in our context.

Replaced URLs with a special token: URLs would otherwise count as one big word that it would be impossible to predict. Instead they are converted into a special token word so that perhaps the model can predict from the context of the sentence that the next word would be an URL. The same was done for emails, user IDs and subreddit IDs.

Split the comments into lines: The LSTM model that will be implemented will have an input dimension of 10, meaning that it will look at the last 10 words in a sentence and try to predict the next word. For this reason the comments were split into lines of 10 words or symbols and the lines which were short of the target length were padded with a special token to keep the sequences of the same length.

Final pass: Any remaining defects in the text that could have arisen from the previous processing steps were scoured such as double white spaces.

Tokenize: The lines were then tokenized with Tensorflow-Keras tokenizer, which assigned every unique word or symbol encountered in the dataset a numerical representation.

Finally, the dataset from LEAF[59] also contains a vocabulary file of the 10,000 most used words in the dataset which was used when tokenizing the text, so that both clients used the same token assignments for the same words. In the case of a word not being in the vocabulary, a special token was assigned to such words.

3.5 Raspberry Pi setup

For this work I had access to two identical Raspberry Pi 4's (4GB RAM). The setup was identical on both devices and followed the following steps:

Set-up latest OS: Using the imager software provided by Raspberry Pi, the latest 64-bit Raspbian operating system (OS) was downloaded and set up on both Raspberry Pi boards. The OS was flashed onto two 32GB mini-SD cards which were then inserted in each Raspberry Pi.

Set-up python packages: After making sure that the OS and pip the package installer for Python were up to date. The necessary packages were installed on each Raspberry Pi board, namely: Flower, Tensorflow, Pandas and Pickle. Flower for the federation, Tensorflow for the deep learning model and pipeline and finally Pandas and Pickle for reading the data and vocabulary file.

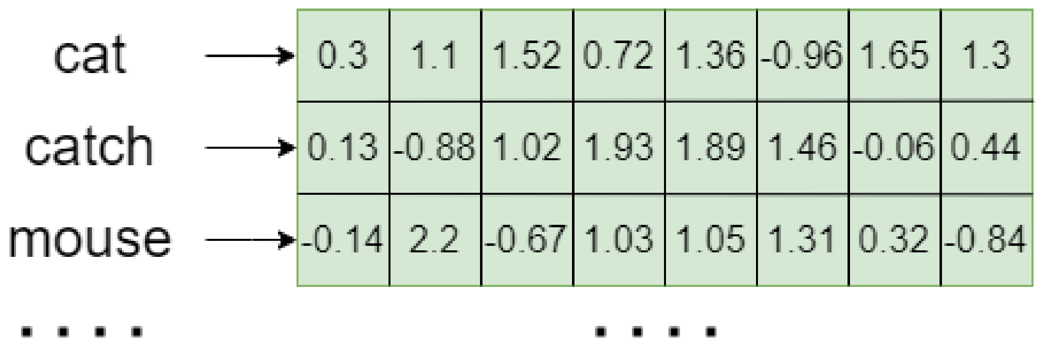
Establish SSH connection: The Raspberry Pi boards were connected to the local machine acting as the server through a SSH connection using the PuTTy SSH terminal. The boards were set up to use the same Wi-Fi connection as the local machine and the IP address of each Raspberry Pi was found and used to set up the connection.

3.6 Overview of the Approach

The deep learning model was implemented using the sequential structure for its simplicity as our implementation is linear and each layer has exactly one input and one output tensor. The model is comprised of 3 layers in the following order:

Embedding Layer

Embedding words allows us to represent them efficiently using a dense representation, where similar words end up with similar encodings. An embedded word is a dense vector of floating point values of a chosen length. The values in an embedding layer are trainable and learnable during the model training. The embedding layer requires specification of an input dimension, which in our case is the size of our vocabulary and an output dimension which is the length of the resulting vector after embedding a word. A larger output dimension allows to capture fine-grained relationships between words but requires more data to properly learn. Finally, in our case we have sequences of 10 words/tokens, so it is necessary to specify the input length to be 10.



cat →	0.3	1.1	1.52	0.72	1.36	-0.96	1.65	1.3
catch →	0.13	-0.88	1.02	1.93	1.89	1.46	-0.06	0.44
mouse →	-0.14	2.2	-0.67	1.03	1.05	1.31	0.32	-0.84
. . . .	. . . .							

Figure 3.3: Diagram showing the embedding process. Each word passing through an embedding layer with output dimension 8, therefore each word converts to a vector of length 8.

LSTM cell layer

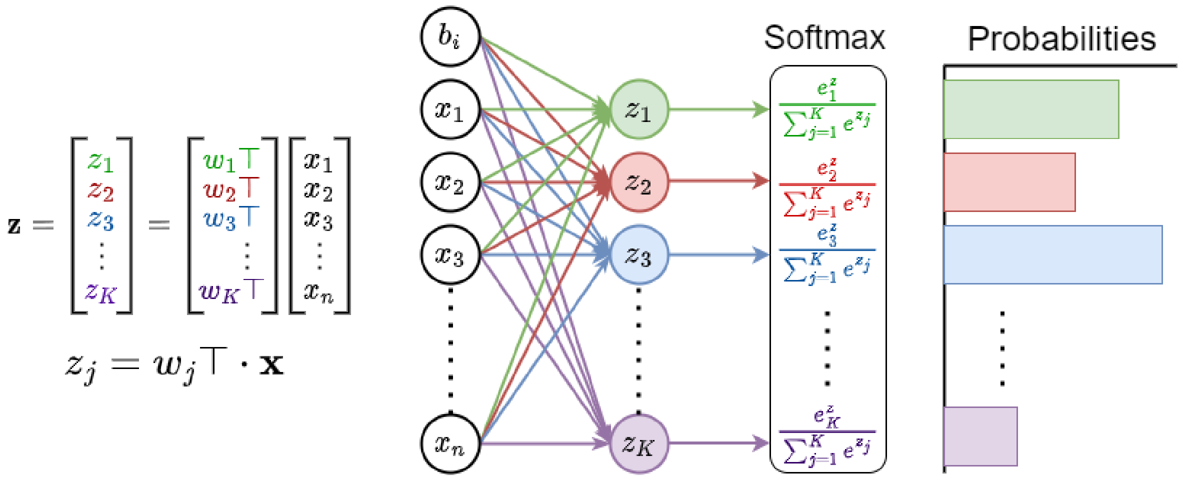
After embedding the sequences we input them to the LSTM network. This layer requires specification of the number of LSTM cell units, which dictates the dimensionality of the output space. Additionally, if we are using Layer Normalization or Recurrent Dropout defined in Section 3.1.1 and 3.1.2, then a LSTM cell from the tfa Tensorflow addon is used instead. The addon version of the LSTM cell is a Layer Normalized version of the standard LSTM cell, which also allows for Recurrent Dropout.

Dense layer

Finally the output of the LSTM cell is fed into a dense layer which is a regular densely connected neural network layer that computes the following: $\mathbf{y} = f(\mathbf{x} * w_x + b_x)$, where f is the element-wise activation function, w_x is the weight matrix of the layer, $\mathbf{x}$ is the input to the layer and b_x is an optional bias vector. This dense layer's input dimension is specified as the size of our vocabulary, and the activation function as softmax so that the output is a vector of probabilities over the vocabulary. The softmax function is defined as:

$$f_{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad (3.11)$$

3.11: Where the probability of a element i of the vector $\mathbf{z}$ is given by the exponential of that element's value e^{z_i} over the sum of the exponential value of all the elements in that vector.



$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} w_1 \top \\ w_2 \top \\ w_3 \top \\ \vdots \\ w_K \top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$
$z_j = w_j \top \cdot \mathbf{x}$
b_i
x_1
x_2
x_3
x_n
z_1
z_2
z_3
z_K
Softmax
$\frac{e_1^z}{\sum_{j=1}^{K} e^{z_j}}$
$\frac{e_2^z}{\sum_{j=1}^{K} e^{z_j}}$
$\frac{e_3^z}{\sum_{j=1}^{K} e^{z_j}}$
$\frac{e_K^z}{\sum_{j=1}^{K} e^{z_j}}$
Probabilities

Figure 3.4: Diagram showing the dense layer with the softmax activation and the resulting probabilities predicting the next word. Where we let the input to the dense layer be $\mathbf{x}$, we denote the vocabulary size as K, the length of the input vector as n, the weights of the dense layer as w_j and a bias term as b_j.

Finally the loss function used for the model is categorical cross-entropy, as next word prediction is a n class classification problem where n is the size of the vocabulary. The categorical cross-entropy is defined as:

$$l_{CE} = -\sum_{i=1}^{n} t_i log(p_i), \quad \text{for n classes} \tag{3.12}$$

3.12: Where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

Illustrated in Figure 3.5 is a high level overview of the deep model pipeline which highlights in green the parameters that will be tuned for the experiments in Chapter 4.

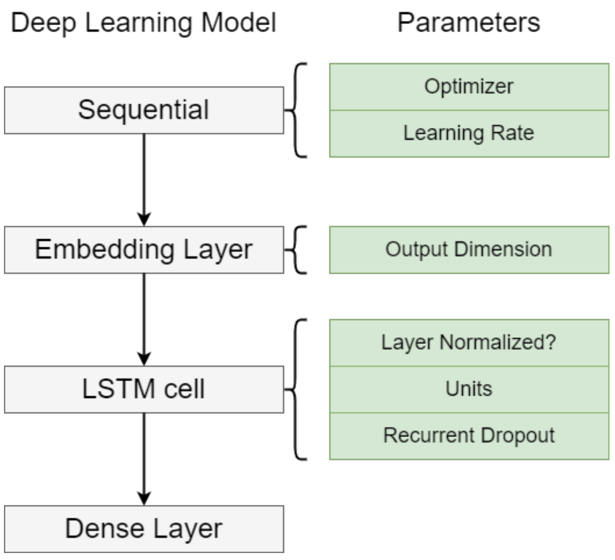


Figure 3.5: Diagram showing the high level structure of the defined sequential model with the tuneable parameters highlighted in green.

Finally, the dataset that was preprocessed in Section 3.4 was partitioned into two sets, with each partition being the comments and posts related to one particular user. The two users chosen were those with the most amount of comments and posts so as to have as much training data as possible as we are limited to only two devices. Each partition was then split into a train set which will be used when training the model and a test set which will be used when evaluating the performance of the global model on that device. Each Raspberry Pi board was pre-loaded with a train file, a test file and the vocabulary file. The small size of the data and preprocessing steps resulted in really small files, sub 200kb which was a desirable outcome as the Raspberry Pi boards are quite storage limited and it wouldn't require much program memory to read these files. Furthermore, each Raspberry Pi was then pre-loaded with a Flower client python script that would be initiated remotely from the local machine through the earlier established SSH connection as illustrated in Figure 3.6. The client and server scripts for FedAvg are illustrated in Appendix B.

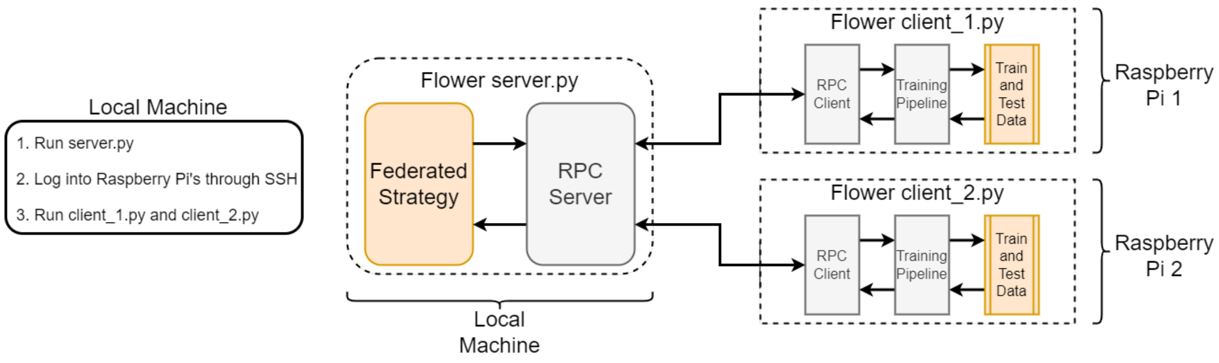


Figure 3.6: Diagram showing the steps in running a FL experiment and the location of the scripts and files.

3.7 Summary

The design chapter established the model specifications and parameters available for tuning that will be used in the experiments in the next chapter. Furthermore, a background and technical introduction for Layer Normalization, Recurrent Dropout and federated optimization algorithms were given. Finally the steps taken to preprocess the data and setup of the experiment were described in detail.

Chapter 4

Evaluation

The evaluation chapter will introduce the experimental set-up and establish a baseline benchmark to compare with the results of implementing Federated Batch Normalization, Federated Proximal and adaptive federated optimizers. The performance metrics of our model will be explained to better understand the effects of each approach.

4.1 Experimental set-up

For the experiments the Raspberry Pi board clients in the FL setting will be referred to as rpi1 and rpi2. Each client was pre-loaded with data sets for training and testing as well as a vocabulary file, the sizes of these files and amount of training and testing sequences are shown in Table 4.1.

Client	Train Data size	Train sequences	Test Data size	Test sequences
rpi1	68kb	1485	28kb	576
rpi2	52kb	1097	16kb	292

Table 4.1: Local data at each client.

As evident by the different sizes of the local data available on each client, we will have not only heterogeneous distribution of data due to each client having the data of a separate user but also heterogeneous data quantity. This means that we will experience parameter shift in our experiments during training. Mitigating this parameter shift is a key goal in this dissertation, however first we need to define a baseline benchmark using Federated Averaging without using any mitigations so that we can compare and judge the efficiency of the different strategies to these results later on.

4.1.1 Performance metrics

As detailed in Section 3.6 the loss function to be minimized in our model is the categorical cross-entorpy, therefore the two metrics which we will pay attention to are accuracy and loss:

Training Loss: The training loss is the loss between the highest probability predicted word after the softmax activation and the ground truth label, given by Equation 3.12. The training loss is computed during the training rounds of the FL process and is used to learn the weights and parameters in the model.

Training Accuracy: The training accuracy is a better visual representation of the performance of the model and is simply a measure of the percentage of correct prediction in a training round.

Testing Loss: The testing loss is similar to the training loss with the difference that it is computed after training, in the testing rounds and its value does not get used to keep learning parameters.

Testing Accuracy: Similarly this measure is calculated during testing instead of training. Testing accuracy serves as a good indicator of how accurate or effective a model is trained and able to predict outside its training data.

4.1.2 Baseline

To establish our baseline benchmark a series of FL runs were performed with different values for the parameters that we can tune. Firstly, the following federated learning parameters were fixed:

Subset of participants: The fraction of participating clients was fixed at 1. This meant that all clients would participate in each training and evaluation round. Normally a smaller fraction of clients would be more practical, but due to only having access to two clients in the experiment it was chosen to do full participation.

Local epochs: The training data available to each client is small so a smaller amount of local computation epochs is preferred. The local epochs were fixed at 1 per training round.

Batch size: The size of the batches used for training was fixed to 32.

Evaluation steps: Lastly it was chosen to perform 5 evaluation steps per evaluation round. This meant, that every evaluation round 5 batches were used for testing the global model on that client's local data.

After fixing the federated learning parameters a short parameter scan was performed to find heuristically optimal values of: the learning rate, LSTM units and embedding output dimension. Plots of the resulting parameter scans are illustrated in Figure 4.1, with the full result tables in the Appendix A. From the parameter scan it is decided to use a

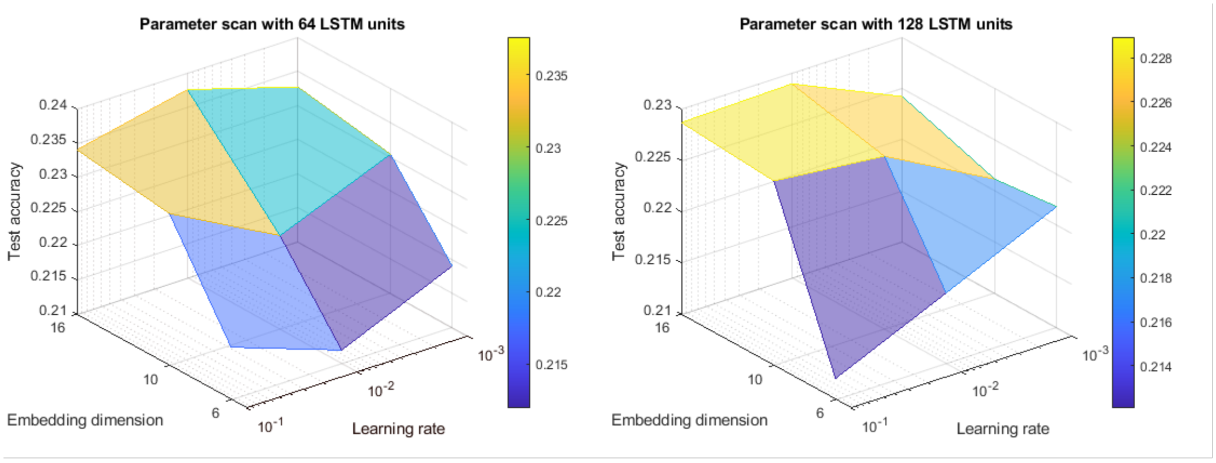


Figure 4.1: Mesh plots showing the parameter scans for 64 and 128 LSTM units.

model with 64 LSTM units, the learning rate $\eta = 0.01$ and the Embedding dimension of 16 as it achieved the highest test accuracy of 23.76%. The test accuracy was computed from the average test accuracy of both clients in the last 10 out of the 50 rounds of federated learning performed. It is important to note that the client rpi2 consistently achieves greater test accuracy than the client rpi1. Illustrated in Figure 4.2 are loss and accuracy plots showing both clients individually. This difference in accuracy is due to

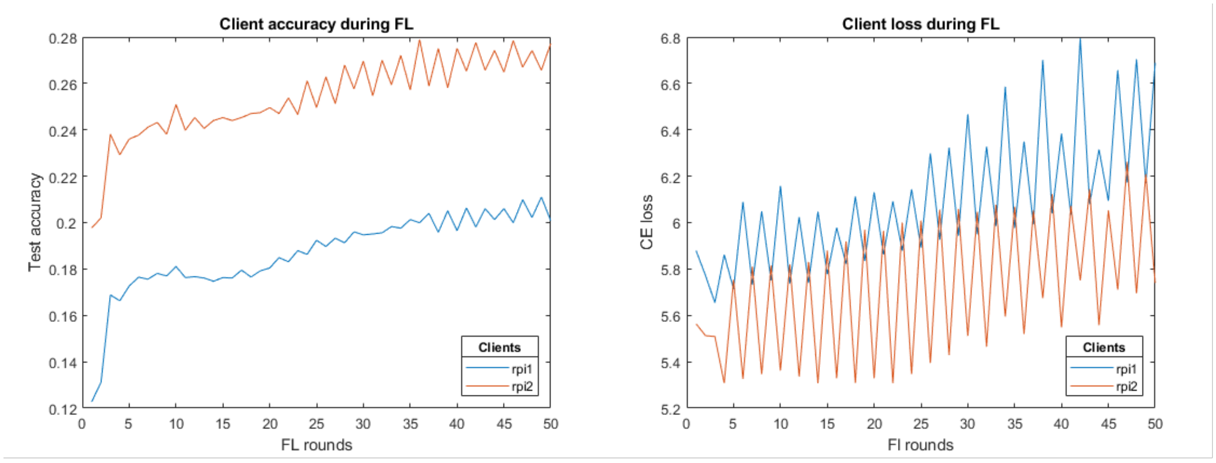


Figure 4.2: Plot showing the test accuracy of the baseline model across the clients.

the client rpi1 having a larger test set, and the model struggles to generalize well enough due to the overall training data being really small. Additionally, the testing loss does not show convergence in 50 rounds and continues increasing. This is once again due to the model not being able to learn the optimal weights due to overfitting the training data. This highlights the need to increase the client pool for the FL system and/or the data available to each client. The baseline model has 830,736 trainable parameters, which is compact enough to run on the Raspberry Pi boards.

Client	Total RAM	% Utilization
rpi1	3.8GB	16%
rpi2	3.8GB	14%

Table 4.2: Memory utilization on Raspberry Pi clients during baseline FL.

4.2 FedProx

As introduced in Section 2.1.3 the optimization algorithm in FedProx is adjusted with a proximal term μ which acts as the penalty for deviation from the global model. In the paper that proposed FedProx [8], the proximal term is set to be chosen heuristically. Therefore, a number of values of μ were selected and tested, and their performances compared to the establised baseline in Figure 4.3. The results show no conclusive improvement or decrease in performance. The middle values of μ in the range of 0.01 to 0.1 show the smoothest testing accuracy curve. However, large values of μ start over-penalizing the updates and exacerbate the jagged testing accuracy curve of the baseline which occurs as a result of both clients trying to reach their model's convergence and oscillating due to the other cleints updates pushing them away from their optimum.

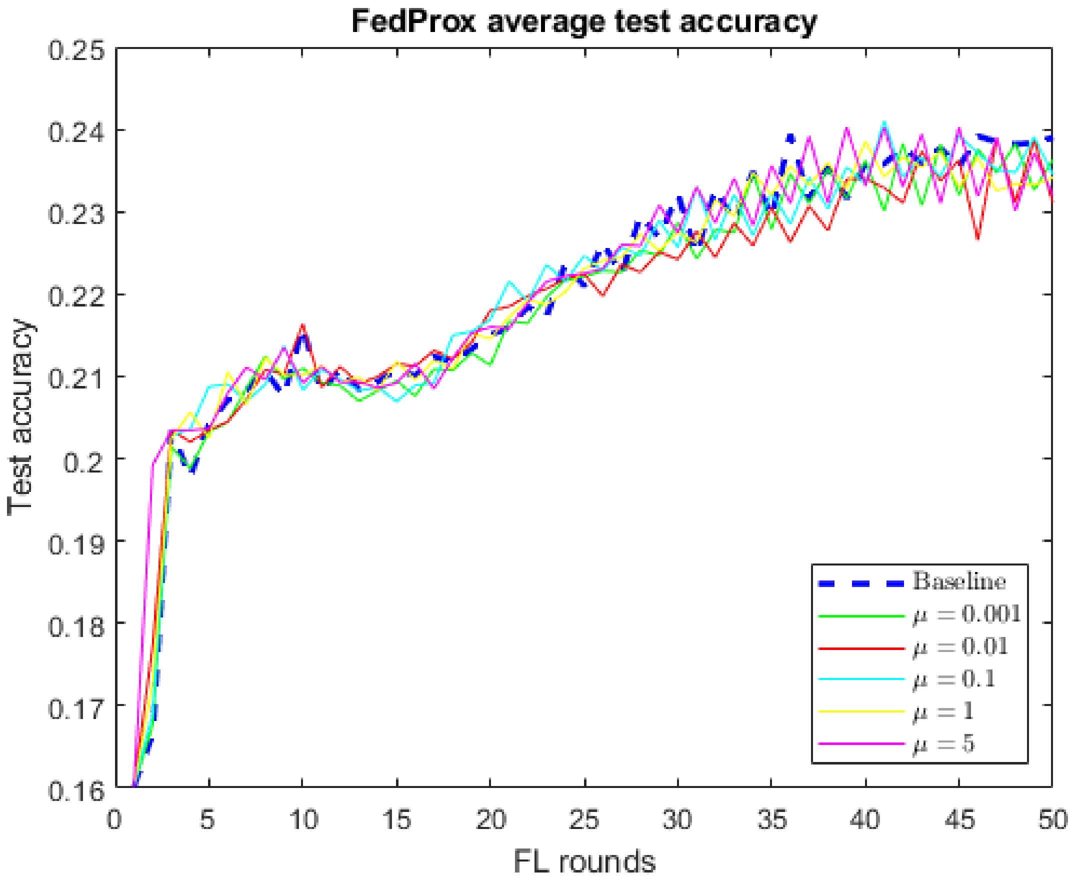


Figure 4.3: Plot showing the test accuracy of different values of the proximal term μ and the baseline FedAvg which corresponds to $\mu = 0$.

4.3 FedBN

FedBN is applied to the clients in the form of Batch Normalization inside the model and therefore the federated strategy remains Federated Averaging. Due to our model being a LSTM network, layer normalization described in Section 3.1.1 is employed. Additionally Recurrent Dropout described in Section 3.1.2 is also employed as it is a common technique to combat overfitting which is present in the baseline model. A number of FL runs were performed using batch normalization and recurrent dropout with different dropout probabilities which are illustrated in Figure 4.4. The results show that FedBN with recurrent dropout leads to significant acceleration in model convergence, with the model converging much faster and achieving a higher maximum testing accuracy but later converging down towards a similar performance as the baseline.

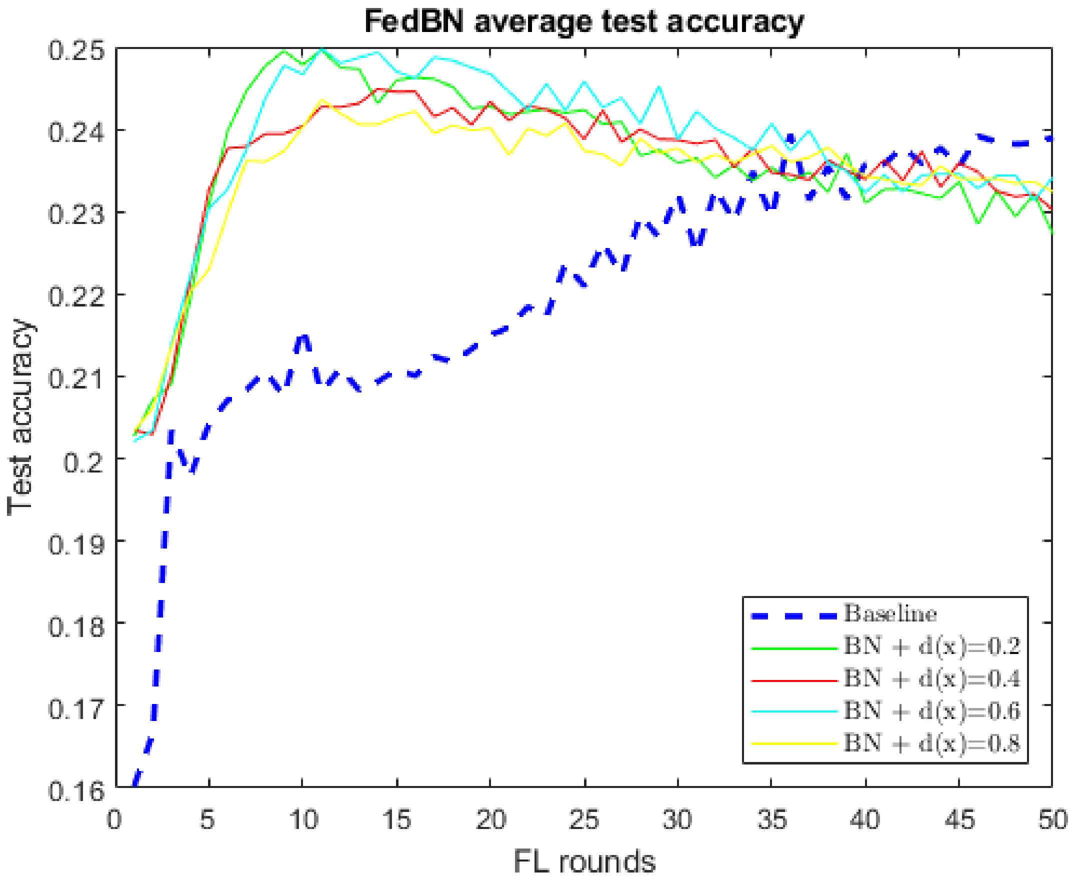


Figure 4.4: Plot showing the test accuracy of different dropout probabilities with layer normalization.

4.4 Adaptive federated optimizers

The federated optimizers FedAdam, FedAdagrad and FedYogi were introduced in Section 3.2. Each federated optimizer was implemented as the federated strategy and compared to the baseline in a similar way as the last two sections. The parameters of the FedAdam, FedAdagrad and FedYogi optimization algorithms were described by the Equation 3.2. For the experiment the parameters were fixed in the following manner: the server and client learning rate were set to equal the learning rate of the baseline implementation $\eta_c = \eta_s = 0.01$, the exponential decay parameters β_1 and β_2 were fixed to 0.9 and 0.99 respectively and finally, the adaptivity parameter τ was set to 1×10^{-9}. These parameters were chosen based on the values chosen in the paper [28] which proposed these federated optimizers.

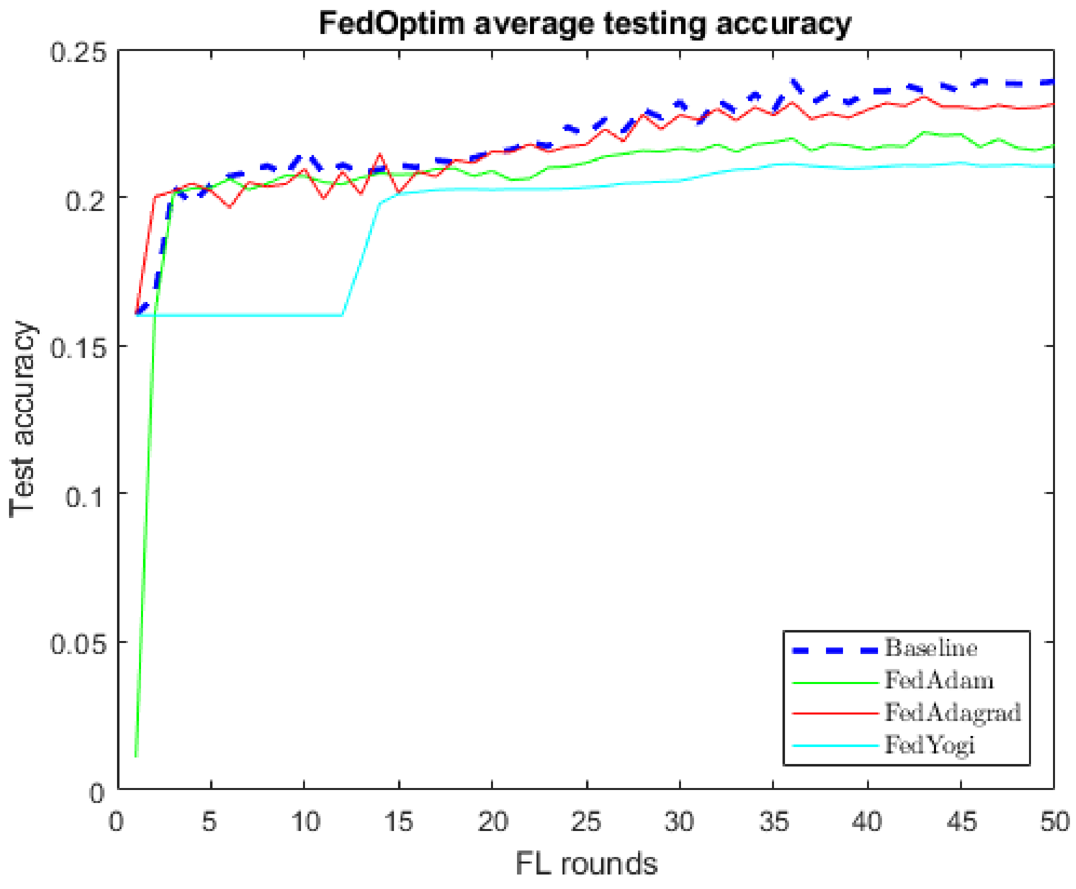


Figure 4.5: Plot showing the test accuracy of different federated optimization algorithms.

The results illustrated in Figure 4.5 show that the performance of the model using federated optimizers is more stable, converging on a slightly lower accuracy but avoiding the jagged shape of the baseline accuracy curve. A more optimal choice of parameters could potentially achieve a higher accuracy, but that it is evident the accuracy is more influenced by the model and data available.

4.5 Summary

The different experiments show the effect of the strategies considered and the reasons for their implementation. However, no significant change in performance is witnessed. The table 4.3 illustrates the average testing accuracy of both clients in the last 10 rounds of the FL process for the best performing configuration of FedBN and FedProx, and the federated optimizers. Notably FedBN with a recurrent probability of 0.6 achieves the highest test accuracy out of all the strategies, and it achieves this accuracy in the first 15 rounds compared to the other approaches which usually peak in the last 10 rounds. It

Experiment	Baseline	FedProx	FedBN	FedAdam	FedAdagrad	FedYogi
Test accuracy	0.2176	0.2316	**0.2338**	0.2185	0.2310	0.2108

Table 4.3: Table of test results.

is evident that the overfitting in the model as a result of the small amount of data and limited amount of clients have a greater influence on the model performance than any possible improvements from the strategies implemented. More detailed result tables of each experiment showing test accuracy and loss are available in the Appendix A.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

This dissertation assessed the feasibility and effectiveness of optimization for heterogeneous data distributions in federated learning next word prediction models on resource-constrained devices such as the Raspberry Pi. Building upon predecessor work on this project, the issue of heterogeneous data distribution and its negative effect on model performance was identified as a key challenge in practical deployments of such FL implementations. Several strategies from literature to tackle this issue were described in detail and implemented to assess their comparative effectiveness in the given task to that of a baseline implementation. A baseline implementation was developed choosing heuristically optimal parameter values from a short non-exhaustive parameter scan. The strategies explored in this dissertation were Federated Batch Normalization, Federated Proximal and three different federated optimizers. While none of the strategies implemented showed any significant improvements in model accuracy, some benefits of each strategy implemented were highlighted, such as convergence speed, stability and addressing overfitting. It was found that the baseline model that was developed suffered from severe overfitting due to the client pool being limited to two Raspberry Pi devices and small datasets used, and that this overfitting problem had more influence on the models performance than data heterogeneity, limiting the potential beneficial impact of the implemented strategies to at most an increase of 1% test accuracy in the experiments carried out. While the work done in this dissertation did not achieve the wanted results significance it hopefully laid the ground work for further experimentation with different strategies on the task of FL next word prediction on resource-constrained devices.

5.2 Limitations and Future Work

Similarly to the conclusions of my predecessors on this project, the key limitation in the work carried out in this dissertation is the number of Raspberry Pi clients available for the FL system. Even though the developed baseline model has different data distributions on the two clients, we fail to see any improvements after implementing mitigation strategies for this issue and the overall accuracy while similar to that of larger scale implementations such as the ones conducted in [28] is still low ($< 30\%$) for practical deployment such as on mobile devices. Therefore, a possible direction for future work on the basis of this dissertation could be deployment of the model on a larger client pool of resource-constrained devices, be it on Raspberry Pi's or mobile devices.

Additionally, the Reddit dataset used while having some favourable qualities such as it being naturally partitioned into users, may not be the most suited for the task of next word prediction. This is reflected with virtually almost no implementations for this task on this dataset, instead using LSTMs for classification by predicting a post's sentiment. The reason for this is most likely due to the typing manner exhibited in the dataset, due to the very informal nature of Reddit, users often type using slang, or shortened versions of words and sometimes even writing the word incorrectly. This in addition with users often typing with symbols makes it very hard for a LSTM to predict the next word without having a large amount of data to train on. For example, the vocabulary used in the implementation in this dissertation was of 10,000 of the most popular words, while both clients together had only 4,332 unique words. This means that more than half of the words that can be predicted by the model are not even encountered once in the training.

Finally, in Section 2.1.2 the challenges facing FL deployments are listed and some notable publications that attempt to address those issues are described. Therefore, another possible direction for future work would be implementing on resource-constrained devices some other method that addresses a different issue faced by FL deployments such as those listed.

Bibliography

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[2] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated learning for mobile keyboard prediction," *arXiv preprint arXiv:1811.03604*, 2018.

[3] N. Rieke, J. Hancox, W. Li, F. Milletari, H. R. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. A. Landman, K. Maier-Hein, *et al.*, "The future of digital health with federated learning," *NPJ digital medicine*, vol. 3, no. 1, p. 119, 2020.

[4] Y. Kumar and R. Singla, "Federated learning systems for healthcare: perspective and recent progress," *Federated Learning Systems: Towards Next-Generation AI*, pp. 141–156, 2021.

[5] J. Xu, B. S. Glicksberg, C. Su, P. Walker, J. Bian, and F. Wang, "Federated learning for healthcare informatics," *Journal of Healthcare Informatics Research*, vol. 5, pp. 1–19, 2021.

[6] G. Long, Y. Tan, J. Jiang, and C. Zhang, "Federated learning for open banking," in *Federated Learning: Privacy and Incentive*, pp. 240–254, Springer, 2020.

[7] Z. Du, C. Wu, T. Yoshinaga, K.-L. A. Yau, Y. Ji, and J. Li, "Federated learning for vehicular internet of things: Recent advances and open issues," *IEEE Open Journal of the Computer Society*, vol. 1, pp. 45–61, 2020.

[8] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[9] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[10] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, *et al.*, "Advances and open problems in federated learning," *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[11] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization," in *International Conference on Artificial Intelligence and Statistics*, pp. 2021–2031, PMLR, 2020.

[12] F. Haddadpour, M. M. Kamani, A. Mokhtari, and M. Mahdavi, "Federated learning with compression: Unified analysis and sharp guarantees," in *International Conference on Artificial Intelligence and Statistics*, pp. 2350–2358, PMLR, 2021.

[13] D. Rothchild, A. Panda, E. Ullah, N. Ivkin, I. Stoica, V. Braverman, J. Gonzalez, and R. Arora, "Fetchsgd: Communication-efficient federated learning with sketching," in *International Conference on Machine Learning*, pp. 8253–8265, PMLR, 2020.

[14] D. A. E. Acar, Y. Zhao, R. M. Navarro, M. Mattina, P. N. Whatmough, and V. Saligrama, "Federated learning based on dynamic regularization," *arXiv preprint arXiv:2111.04263*, 2021.

[15] T. Li, M. Sanjabi, A. Beirami, and V. Smith, "Fair resource allocation in federated learning," *arXiv preprint arXiv:1905.10497*, 2019.

[16] A. Fallah, A. Mokhtari, and A. Ozdaglar, "Personalized federated learning with theoretical guarantees: A model-agnostic meta-learning approach," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3557–3568, 2020.

[17] Z. Yang and Q. Sun, "A dynamic global backbone updating for communication-efficient personalised federated learning," *Connection Science*, vol. 34, no. 1, pp. 2240–2264, 2022.

[18] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous online federated learning for edge devices with non-iid data," in *2020 IEEE International Conference on Big Data (Big Data)*, pp. 15–24, IEEE, 2020.

[19] Z. Xu, F. Yu, J. Xiong, and X. Chen, "Helios: Heterogeneity-aware federated learning with dynamically balanced collaboration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 997–1002, IEEE, 2021.

[20] J. Park, D.-J. Han, M. Choi, and J. Moon, "Sageflow: Robust federated learning against both stragglers and adversaries," *Advances in Neural Information Processing Systems*, vol. 34, pp. 840–851, 2021.

[21] S. Caldas, J. Konečny, H. B. McMahan, and A. Talwalkar, "Expanding the reach of federated learning by reducing client resource requirements," *arXiv preprint arXiv:1812.07210*, 2018.

[22] C. He, M. Annavaram, and S. Avestimehr, "Group knowledge transfer: Federated learning of large cnns at the edge," *Advances in Neural Information Processing Systems*, vol. 33, pp. 14068–14080, 2020.

[23] M. Rapp, R. Khalili, and J. Henkel, "Distributed learning on heterogeneous resource-constrained devices," *arXiv preprint arXiv:2006.05403*, 2020.

[24] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas, "Model pruning enables efficient federated learning on edge devices," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[25] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "Splitfed: When federated learning meets split learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, pp. 8485–8493, 2022.

[26] X. Li, M. Jiang, X. Zhang, M. Kamp, and Q. Dou, "Fedbn: Federated learning on non-iid features via local batch normalization," *arXiv preprint arXiv:2102.07623*, 2021.

[27] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 429–450, 2020.

[28] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečnỳ, S. Kumar, and H. B. McMahan, "Adaptive federated optimization," *arXiv preprint arXiv:2003.00295*, 2020.

[29] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, pmlr, 2015.

[30] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[31] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar, "Adaptive methods for nonconvex optimization," *Advances in neural information processing systems*, vol. 31, 2018.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955," *AI magazine*, vol. 27, no. 4, pp. 12–12, 2006.

[34] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[35] J. Allen, *Natural Language Understanding.* USA: Benjamin-Cummings Publishing Co., Inc., 1988.

[36] C. Manning and H. Schutze, *Foundations of statistical natural language processing.* MIT press, 1999.

[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[38] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[40] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[41] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[42] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[43] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[44] F. Gers and E. Schmidhuber, “Lstm recurrent networks learn simple context-free and context-sensitive languages,” *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1333–1340, 2001.

[45] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[46] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan, *et al.*, “Towards federated learning at scale: System design,” *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[47] T. T. F. Authors, “TensorFlow Federated,” 12 2018.

[48] A. Ziller, A. Trask, A. Lopardo, B. Szymkow, B. Wagner, E. Bluemke, J.-M. Nounahon, J. Passerat-Palmbach, K. Prakash, N. Rose, T. Ryffel, Z. N. Reza, and G. Kaissis, “Pysyft: A library for easy federated learning,” 2021.

[49] C. He, S. Li, J. So, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr, “Fedml: A research library and benchmark for federated machine learning,” *Advances in Neural Information Processing Systems, Best Paper Award at Federate Learning Workshop*, 2020.

[50] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, K. H. Li, T. Parcollet, P. P. B. de Gusmão, *et al.*, “Flower: A friendly federated learning framework,” 2022.

[51] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” *arXiv preprint arXiv:1607.06450*, 2016.

[52] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv preprint arXiv:1207.0580*, 2012.

[53] T. Moon, H. Choi, H. Lee, and I. Song, “Rnndrop: A novel dropout for rnns in asr,” in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pp. 65–70, IEEE, 2015.

[54] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[55] S. Semeniuta, A. Severyn, and E. Barth, "Recurrent dropout without memory loss," *arXiv preprint arXiv:1603.05118*, 2016.

[56] D. Gadler, "Federated learning of a recurrent neural network on raspberry pis," May 2019.

[57] A. Mathur, D. J. Beutel, P. P. B. de Gusmao, J. Fernandez-Marques, T. Topal, X. Qiu, T. Parcollet, Y. Gao, and N. D. Lane, "On-device federated learning with flower," *arXiv preprint arXiv:2104.03042*, 2021.

[58] "Reddit."

[59] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečnỳ, H. B. McMahan, V. Smith, and A. Talwalkar, "Leaf: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018.

[60] J. Baumgartner, S. Zannettou, B. Keegan, M. Squire, and J. Blackburn, "The pushshift reddit dataset," in *Proceedings of the international AAAI conference on web and social media*, vol. 14, pp. 830–839, 2020.

Appendix A

Appendix

Parameters	$od = 6$	$od = 10$	$od = 16$
$\eta = 0.001$	0.2218	0.2211	0.2242
$\eta = 0.01$	0.2169	0.2280	0.2287
$\eta = 0.1$	0.2121	0.2280	0.2887

Table A.1: Accuracy parameter scan with fixed 128 LSTM units.

Parameters	$od = 6$	$od = 10$	$od = 16$
$\eta = 0.001$	6.4490	6.4743	6.3977
$\eta = 0.01$	6.4841	6.4226	6.3117
$\eta = 0.1$	6.4841	6.3884	6.3851

Table A.2: Loss parameter scan with fixed 128 LSTM units.

Parameters	$od = 6$	$od = 10$	$od = 16$
$\eta = 0.001$	0.2190	0.2304	0.2327
$\eta = 0.01$	0.2120	0.2237	**0.2376**
$\eta = 0.1$	0.2177	0.2322	0.2341

Table A.3: Accuracy parameter scan with fixed 64 LSTM units.

Parameters	$od = 6$	$od = 10$	$od = 16$
$\eta = 0.001$	6.1698	6.0806	6.1392
$\eta = 0.01$	6.0649	6.0741	**6.1452**
$\eta = 0.1$	6.0649	6.1429	6.1151

Table A.4: Loss parameter scan with fixed 64 LSTM units.

Parameters	$\mu = 0.001$	$\mu = 0.01$	$\mu = 0.1$	$\mu = 1$	$\mu = 5$
Loss	6.1604	6.0929	6.1783	6.1183	6.1714
Accuracy	0.2349	0.2338	0.2366	0.2346	0.2354

Table A.5: Test results for different proximal term values.

Parameters	$d(x) = 0.2$	$d(x) = 0.2$	$d(x) = 0.2$	$d(x) = 0.2$
Loss	6.7723	6.8241	6.7387	6.7256
Accuracy	0.2313	0.2338	0.2338	0.2338

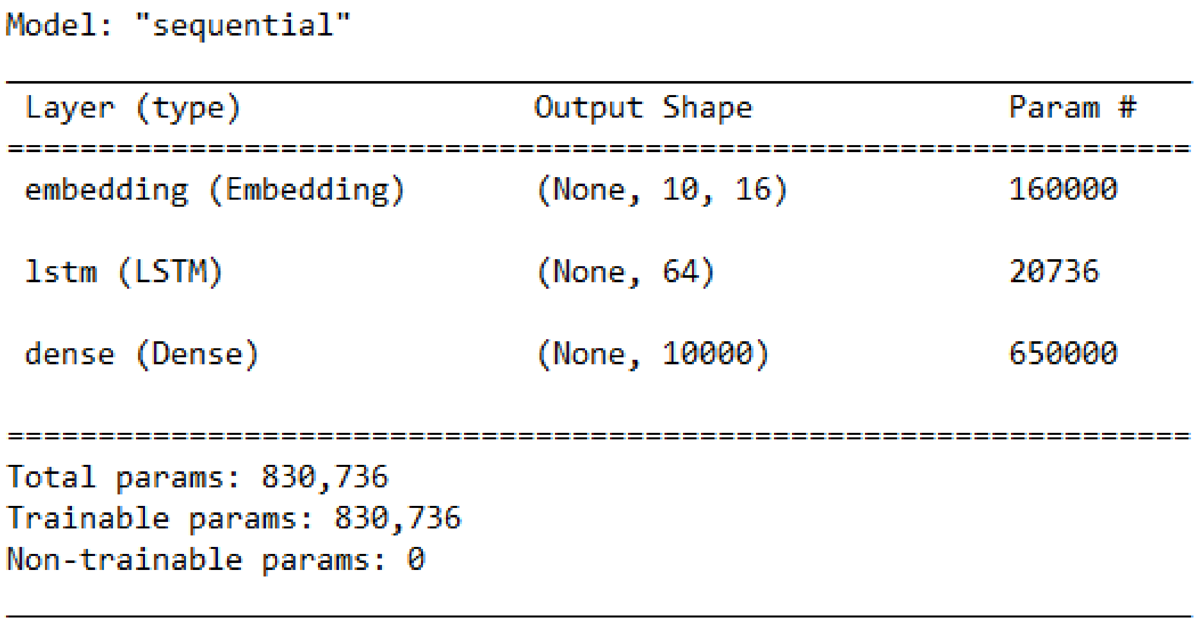
Table A.6: Test results for different dropout probabilities with layer normalization.

Parameters	FedAdagrad	FedAdam	FedYogi
Loss	5.9456	6.312	5.5615
Accuracy	0.2310	0.2185	0.2108

Table A.7: Test results for different federated optimizers.

Appendix B

Appendix



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 10, 16)            160000

 lstm (LSTM)                 (None, 64)                20736

 dense (Dense)               (None, 10000)             650000

=================================================================
Total params: 830,736
Trainable params: 830,736
Non-trainable params: 0
_________________________________________________________________
```

Figure B.1: Baseline model structure with parameter count.

```
import dependencies

def main() -> None:
    # Define model
    model = Sequential()
    model.add(Embedding(input_dim=10000, output_dim= od, input_length=10))
    model.add(LSTM(units))
    model.add(Dense(10000, activation='softmax'))
    optimizer = Adam(learning_rate= η)
    model.compile(loss = 'categorical_crossentropy', optimizer='adam',
        metrics=["accuracy"])
    # Create strategy
    strategy = fl.server.strategy.FedAvg(
        # Subset C of clients selected, i.e. C = 1 is full praticipation
        fraction_fit=1,
        min_fit_clients=2,
        min_available_clients=2,
        on_fit_config_fn=fit_config,
        on_evaluate_config_fn=evaluate_config,
        # Initialize model parameters at random
        initial_parameters=model.get_weights(),
    )

    # Start Flower server
    history = fl.server.start_server(
        server_address="[::]:8080",
        # Set the number of rounds to run FL
        config=fl.server.ServerConfig(num_rounds),
    )

# Return a training configuration dict for each round
def fit_config():
    config = {
        # Keep batch size fixed at 32
        "batch_size": 32,
        # Keep local epochs fixed at 1
        "local_epochs": 1,
    }
    return config

# Return an evaluation configuration dict for each round
def evaluate_config():
    # Keep validation steps fixed at 5
    val_steps = 5
    return {"val_steps": val_steps}


if __name__ == "__main__":
    main()
```

Listing B.1: Flower server script skeleton.

```
import dependencies

# Define Flower client
class CifarClient(fl.client.NumPyClient):
    def __init__(self, model, x_train, y_train, x_test, y_test):
        self.model = model
        self.x_train, self.y_train = x_train, y_train
        self.x_test, self.y_test = x_test, y_test

        # Get parameters from server
    def get_parameters(self, config):
        return self.model.get_weights()

     # Fit model using fit_config
    def fit(self, parameters, config):
        self.model.set_weights(parameters)
        history = self.model.fit(self.x_train, self.y_train, batch_size,
            epochs, shuffle=True)

        # Return updated model parameters and results
        parameters_prime = self.model.get_weights()
        return parameters_prime

    # Evaluate model using evaluate_config
    def evaluate(self, parameters, config):
        self.model.set_weights(parameters)
        loss, accuracy = self.model.evaluate(self.x_test, self.y_test,
            steps=val_steps)
        return loss, accuracy


def main() -> None:
    # Load train, test and vocabulary files
    train_data = pd.read_csv('train_path')
    test_data = pd.read_csv('test_path')
    vocab_file = pickle.load(open('vocab_path', 'rb'))
    vocab = collections.defaultdict(lambda: vocab_file['unk_symbol'])
    vocab.update(vocab_file['vocab'])

    # Tokenize the vocab to create index and use oov for unkonwn words
    tokenizer = Tokenizer(num_words=vocab_file['size'], lower=True,
        oov_token='<oov>')
    tokenizer.fit_on_texts(vocab)
    total_words = vocab_file['size']

    # Create train and test sequences
    input_sequences = []
    for line in train_data['comments']:
```

```
    token_list = tokenizer.texts_to_sequences([line])[0]

    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

test_sequences = []
for test_line in test_data['comments']:
    test_token_list = tokenizer.texts_to_sequences([test_line])[0]

    for j in range(1, len(test_token_list)):
        test_n_gram_sequence = test_token_list[:j+1]
        test_sequences.append(test_n_gram_sequence)

# Pad sequences
max_sequence_len = 11
input_sequences = np.array(pad_sequences(input_sequences,
   maxlen=max_sequence_len, padding='pre'))
test_sequences = np.array(pad_sequences(test_sequences,
   maxlen=max_sequence_len, padding='pre'))

xs, slabels = input_sequences[:,:-1],input_sequences[:,-1]
ys = tf.keras.utils.to_categorical(slabels, num_classes=total_words)

x, labels = test_sequences[:,:-1],test_sequences[:,-1]
y = tf.keras.utils.to_categorical(labels, num_classes=total_words)



# Load and compile model
model = Sequential()
model.add(Embedding(input_dim=10000, output_dim= od, input_length=10))
model.add(LSTM(units))
model.add(Dense(10000, activation='softmax'))
optimizer = Adam(learning_rate= η)
model.compile(loss = 'categorical_crossentropy', optimizer='adam',
    metrics=["accuracy"])

# Call client
client = CifarClient(model, xs, ys, x, y)

# Connect to server
fl.client.start_numpy_client(
    server_address="127.0.0.1:8080",
            client=client,
    )

if __name__ == "__main__":
    main()
```

Listing B.2: Flower client script skeleton.