Federated and Tiny Machine Learning for Edge Computing in IoT

Enabling microcontrollers' viability in Federated Learning

Vegard O. Årnes

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2023

Federated and Tiny Machine Learning for Edge Computing in IoT

Enabling microcontrollers' viability in Federated Learning

Vegard O. Årnes

Federated and Tiny Machine Learning for Edge Computing in IoT

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

With the emergence of IoT and microcontrollers in general, as well as advancements in machine learning processes, the desire to continuously automate and process the world on smaller and smaller mediums has grown with the shrinking of computational power. Known libraries exist for developing ML models for inference on devices, however, the act of decentralizing the entire training process to devices is still in its infancy. As such this task seeks to address the potential of deploying such machine learning capability on microcontrollers and explore what improvements can be gained by leveraging federated learning to have small devices cooperate in creating a large enough, and sufficiently accurate mode for different use cases. The thesis seeks to test that primarily by running on-device LSTM model training on PM10.0 data gathered by NILU, however briefly explores the potentials of on-device training of DNNs, and CNNs for image classification specifically.

Acknowledgements

This thesis has been submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements to obtain the degree of Masters of Sciences. The studies presented were carried out between Autumn 2022 and spring 2023. My supervisors have been Associate Professor Amir Taherkordi and Dr. Dapeng Lan, whom I would like to extend my greatest gratitude for their invaluable patience and feedback during this process.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview

By leveraging advancements in artificial intelligence (AI) and cloud computing as well as in microcontrollers and the Internet of Things (IoT), industry 4.0 has exponentially improved operational and manufacturing processes. Practical application of edge IoT in particular has been improved by having the data provided by sensors more accurately interpreted by machine learning (ML)-driven cloud service deployments. Decisions thereafter dictate the behavior of actuator microcontrollers or can serve other analytical purposes. Automation of complex tasks using such IoT nodes has naturally shown its benefits in terms of efficiency, ultimately resulting in monetary gain for adopters of the technology. However, while the concept has proven beneficial for many such operational and manufacturing processes, the practical application of the technology is still in its infancy. As such, a number of issues and areas in need of improvement have been identified. Relevant issues include those related to data privacy as centralized storage of data has proven to be vulnerable to leaks, in addition to the issue of lacking training data necessary for achieving viable AI models.

In the case of IoT devices controlled using centralized artificial intelligence (AI), standard issues suffered include the lack of data set necessary for viable model generation, as well as the latency between sensors, cloud service, and edge device, as well as the privacy implications involved in traditional centralized deployments[29]. Therefore, advancement in existing services can be achieved by improving existing processes or changing the overall architecture of existing implementation to eliminate the issues entirely.

1.2 Research Questions

The project will revolve around creating a test bed for deploying federated learning on resource-constrained devices. As available libraries for inference do not provide functionality for training on microcontrollers, the first and foremost requirement is creating a system that allows for efficient on-device training of common types of neural networks. The three types of neural networks explored for on-device training are deep neural networks, Convolutional neural networks, and recurrent neural networs, focusing in particular on long short-term memory. These were chosen as they cover several ML use cases well. Finally, FL will be explored as the means of training the models, and aspects of model compression before and during FL, as well as data distribution, number of participants, and other variables, will be evaluated to find the optimal setup. In the end, parameters for on-device training and the FL context and training parameters will be assessed to conclude the viability of tiny and federated ML, and how to best deploy it.

Because of the constraints faced by microcontrollers in the form of limited storage and computational ability, the extent to which fully trainable models can be deployed on modern IoT devices is in need of this exploration

as sacrifices in terms of data quantity, model size, number of parameters, and data types might be needed to obtain the best possible outcome. This is of course necessary as the model and data would have to fit on the device, but it is also especially necessary as fitting the model in SRAM significantly reduces power usage[10] and increases computational speed by not needing to access parameters in flash memory. With the emergence of federated learning, privacy has become more secure by removing the need for the transferal and central storage of data[18], the added benefit however is the access to more data through collaborative effort and potentially having varying parameters. As such, exploring the potential of offsetting potential on-device training optimization trade-offs through federated learning is necessary to comprehend the limits and potential of ML on microcontrollers.
Therefore the research questions are as follows:

1. Can sufficiently accurate models be generated from distributed machine learning on resource-constrained devices
2. What are the potential benefits and drawbacks of combining federated learning and tiny machine learning?
3. What factors impact the system performance for using federated learning in IoT applications and how can the system performance be optimized?

1.3 Methodology

The research starts with a literature study to understand existing work on model compression, Federated Learning, and Tiny machine learning, respectively, and together to understand the state of the art in FL using microcontrollers. Thereafter the framework for on-device training was developed in an iterative process as meetings with the supervisor were held roughly bi-weekly, which served as points to re-evaluate the path forward. In order to prioritize the order of tasks to complete, a simple scrum board was utilized to keep track of what had to be done and at what point functionalities were in the development. As the development of the project started, certain focuses changed. Due to the lack of on-device training libraries for microcontrollers, the priority of the thesis shifted to developing that after having explored the existing libraries which are used only for inference, and realizing their limitations. As such creating an extensive test bed for on-device training for microcontrollers compatible with federated learning became the core focus.

The data used during training of the ML algorithms include secondary sourced real-world data from NILU for testing of LSTM training, while CIFAR-10 was used for testing of DNN and CNN training. The federated learning process was implemented and tested by running the on-device training on several Arduino and Sparkfun devices connected to the computer through the serial port with the FL happening through a Python script working as a server. For the sake of learning, simulations were also run using the same code, with an excessive number of devices with various

distributions of data to acquire knowledge unobtainable with the physical test bed.

In order to evaluate the performance of the models tested, various common metrics for evaluating neural networks are employed. LSTMs for predicting digits on time series data use the Mean Squared Error loss function, and Mean Average Percentage Error as metrics for understanding the accuracy. DNN and CNN both rely on accuracy, F1 score, precision and recall to gain an understanding of their performance. Finally, federated learning uses the same metrics for garnering an understanding of the model performance after each weight averaging. However, as it is decentralized, other metrics are needed to evaluate the efficiency of the communication. As such, metrics regarding communication cost and convergence statistics are based on the number of federated learning rounds and local episodes. Thereafter, understanding the implications on power consumption is necessary to estimate the actual real-world applicability of the neural networks as both model size and computation complexity can be represented in the power consumption of the microcontroller. The performance of the model can further be understood when taking the power into account.

1.4 Contribution

1. A library bringing machine learning capabilities to resource-constrained IoT nodes
2. Establish reasons for why federated learning in the context of IoT can be preferable to centralized learning with an edge-based model
3. Illuminate potential trade-offs to costs, performance, and accuracy when optimizing aspects of on-device training

1.5 Structure of Thesis

Chapter 1: Introduction The introduction explains the purpose of the thesis and a general overview, highlighting the research questions, explaining the methodology, and listing the contributions.

Chapter 2: Background The background chapter covers essential theory in relation to ML and FL, as well as explains aspects of microcontrollers in general and their viability for use in complex computation. Thereafter, existing works on similar topics both regarding on-device training alone and FL in general are assessed. Finally, the overall situation of FL and ML on microcontrollers is discussed, addressing requirements for its viability.

Chapter 3: Related Works An analysis of related works regarding on-device training of machine learning models for microcontrollers, general state-of-the-art training strategies in the context of both centralized and federated machine learning, and finally, compression methods for machine learning models are addressed.

Chapter 3: Proposed Framework The implementation is described to illustrate what was done to achieve the results of the thesis. Reasons for choices regarding implementation, neural networks used and their parameters, and choices in algorithms like compression are explained. Finally, the unaddressed work is summarised.

Chapter 4: Evaluation The chapter covers the tests done and results generated, showing different effects of adjustments made to the various parameters involved in the training of neural networks and in different FL contexts. Based on the data gathered, a final discussion and evaluation of the viability of tiny ML in conjunction with Federated learning for training sufficiently accurate models for common ML problems, using DNN, CNN and LSTM to test image recognition and time series prediction will be done for to further extract information from the results.

Chapter 6: Conclusion & Future Work Finally, the thesis will conclude with an explanation of the selected best approach to the scenarios, then address the parameters for on-device and federated learning respectively, and their effect on performance and quality, in addition to a guideline to potential trade-offs possible for gains in specific areas. Then in the end insight will be provided into future work which would be beneficial to pursue after the thesis, particularly covering ways of extending this work, addressing weaknesses present, and other strategies for training which should be addressed.

2 Background

In order to address the specific issues related to the on-device deployment of machine learning and the potential of its use together with Federated learning, some fundamentals need to be understood. Technical aspects of machine learning are briefly explained to provide an understanding of the architectures to be deployed on the devices, as well as alternatives in parameters, what potential benefits they pose, and the potential costs of choosing them. Secondly, the known constraints of microcontrollers are addressed, and known optimization strategies for decreasing ML model sizes to alleviate these issues are described, as well as methods for reducing the quality loss of deploying such optimization strategies. Finally, the state of training paradigms like centralized learning and federated learning are presented, highlighting features that might make them more or less desirable based on the deployment circumstance, focusing on the use-case of deployment for microcontrollers. Additional information like relevant technologies, evaluation metrics, and common microcontrollers are also briefly addressed for a complete overview of the fundamentals crucial to following the approach in the thesis.

2.1 Artificial Neural Networks

An artificial neural network is a computational model inspired by the structure and function of biological neural networks, which are the basic building blocks of the brain. It is a powerful tool for solving complex problems in various fields, including pattern recognition, image and speech recognition, natural language processing, robotics, and control systems. The structure of an artificial neural network consists of layers of interconnected nodes, called neurons, which process and transmit information. Each neuron receives input from multiple other neurons, performs a nonlinear transformation of the input, and produces an output that is transmitted to other neurons in the next layer. The connections between neurons are weighted, which allows the network to learn from data by adjusting the weights to minimize a loss function. Training an artificial neural network involves feeding it with a set of input-output pairs and adjusting the weights of the connections to minimize the difference between the predicted and actual outputs. This process is typically done using a variant of the stochastic gradient descent algorithm, which updates the weights based on the gradient of the loss function with respect to the weights.

2.1.1 Deep Neural Networks

A deep neural network is a specific type of ANN which is built up of hidden layers between the input and output layers. The hidden layers consist of weights and a bias where each neuron is connected to the former and latter neuron layers. As a result, the number of parameters is large relative to the training data and may result in overfitting due to the complexity as it memorizes training data instead of patterns. It also has a black-box nature due to the difficulty of understanding how it reaches its conclusion. However,

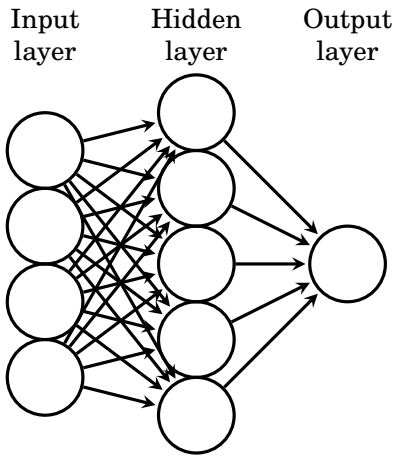


Figure 1: A simple DNN with input size of 4, 5 nodes in hidden layer, and single output node

due to the simple architectural design and computation, it is a relatively reliable and easy way of creating an ANN.

2.1.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of deep neural network that is commonly used for image classification, object detection, and other computer vision tasks. A CNN is designed to recognize patterns and features in images by using a special type of layer called a convolutional layer. This layer applies a set of filters to the input image, which extracts features such as edges, lines, and corners. The output of the convolutional layer is then passed through a non-linear activation function, such as ReLU (Rectified Linear Unit), to introduce non-linearity into the model. The output of the convolutional layers is then passed through one or more fully connected layers, which perform the final classification task. The fully connected layers take the output of the convolutional layers and transform it into a prediction for each class in the classification task. CNNs have been very successful in image classification tasks, achieving state-of-the-art results on large datasets such as ImageNet. They have also been used for a variety of other tasks, including object detection, image segmentation, and even natural language processing.

2.1.3 Recurrent Neural Network: Long Short-Term Memory

LSTM stands for ”Long Short-Term Memory,” and it is a type of recurrent neural network (RNN) that is designed to handle the problem of vanishing gradients that can occur in traditional RNNs[13]. An LSTM network includes a memory cell that can store information over long periods of time. There are three main gates that control the memory cell, the input gate, the output gate, and the forget gate. These gates control the flow of in-

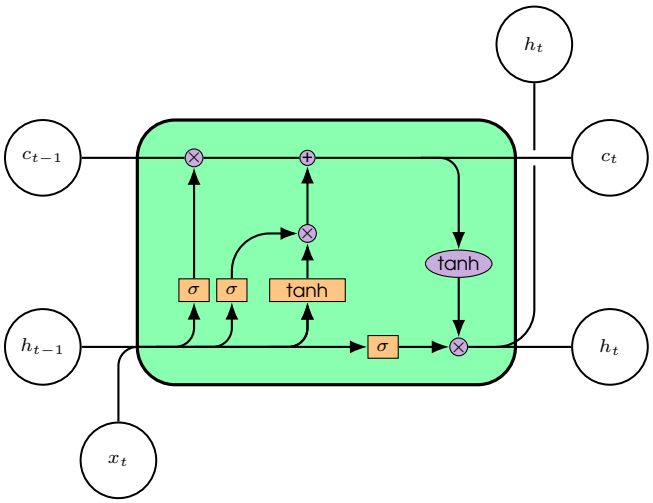


Figure 2: Basic design of an LSTM cell

formation in and out of the memory cell and give the network the ability to selectively forget or remember information based on what is needed. LSTM networks have been successful in a vast variety of applications, including speech recognition, language modeling, and machine translation. In addition, they are particularly well-suited for tasks that involve sequences of data, such as predicting the next word in a sentence or the next frame in a video, or the next number in a sequence. The training process of an LSTM typically involves using backpropagation through time (BPTT), a variant of backpropagation that takes into account the temporal nature of the data. The way BPTT works is by unrolling the network through time and calculating gradients at each time step, which are then used to update the network's parameters when the backpropagation takes place.

2.2 Activation Functions

Activation functions are important parts of neural networks. They can determine the output of a node when given input and can be applied in different scenarios based on their properties. Their practical effect can range from assisting in convergence speed to allowing the network to be trained in the first place. Some activation functions like Sigmoid and softmax are more commonly applied on the output layer as Sigmoid works well for binary and multilabel classification due to the input resulting in a certainty percentage for every output, while softmax better handles multiclass classification. Meanwhile, other activation functions are more commonly seen applied to the hidden neurons like ReLu in CNN for assisting complex pattern recognition and Tanh in the output gate in LSTM for determining the output from the cell state which can be any real number.

2.2.1 Sigmoid

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

The sigmoid activation function is a mathematical function that is often employed as the non-linear activation function in neural networks. It maps any input value to an output value between 0 and 1 based on an S-shaped curve, making it particularly useful in applications where the output needs to be constrained within a specific range. It is often used in neural networks to introduce non-linearity into the model, allowing it to learn and represent more complex relationships between the inputs and outputs. Specifically, the sigmoid function is typically applied to the output of each neuron in a neural network, which allows the neuron to produce an output that can take on continuous values between 0 and 1. As an example, if the input is close to negative infinity, the sigmoid value approaches 0, if input is 0 sigmoid returns 0.5, and digits close to infinity return 1. This allows the network to learn more complex decision boundaries and produce more nuanced outputs. Despite its viability, the Sigmoid function has some drawbacks, particularly with respect to vanishing gradients, which can occur when the input to the function is very large or small. This can make training neural networks with Sigmoid activation functions challenging, particularly in deep neural networks where the gradient can become very small as it is propagated back through the layers. This can however be mitigated to some extent with careful tuning and regularization.

2.2.2 Softmax

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2}$$

The softmax activation function is commonly used in neural networks for multiclass classification problems. It is a generalization of the sigmoid activation function and maps a K-dimensional vector of real values to a similar dimension vector of values between 0 and 1 that add up to 1. It is often used as the final activation function in a neural network for multiclass classification problems. Specifically, the output of the network's last layer is fed through the softmax function, which produces a probability distribution over the possible classes. The class with the highest probability is then selected as the predicted class for the input. The softmax function is particularly useful for problems where the output needs to be a probability distribution over a set of classes. It is also differentiable, which makes it suitable for use in backpropagation-based learning algorithms, such as stochastic gradient descent. However, a drawback that softmax has is that it can suffer from numerical instability when the input values are very large or very small, these issues can also, like with sigmoid, be mitigated through regularization techniques and careful implementation.

2.2.3 Rectified Linear Unit

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

The Rectified Linear Unit (ReLU) activation function is widely used non-linear activation function and is a simple linear function that returns the input value if it is positive, and zero if it is not. It has several advantages over other activation functions. First, it is computationally efficient to compute, second, it promotes sparsity in neural networks, which helps in preventing overfitting and improve generalization performance. Third, it does not suffer from the vanishing gradient problem in other functions like sigmoid and tanh. It also has drawbacks however, with one being the "dying ReLU" problem, which can occur when a large portion of the neurons in a network become output zero for all inputs, effectively becoming "dead". An example of when this dying ReLu problem can happen is when the weights of the neurons are initialized in such a way that they push the neurons into the zero region of the function, and the training process doesn't help the neurons recover. Alternatives to standard ReLu to address this problem have been made, i.e. leaky ReLU, where a small non-zero output for negative input values is allowed, and the exponential ReLU, where the transition between the zero and non-zero regions is smoother.

2.2.4 Tanh

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4}$$

2.3 Gradient descent algorithms

The gradient descent algorithm utilized by a model serves as a core feature in how the training process is handled, as it provides an essential mechanism for learning from data. These algorithms are optimization procedures, designed with the purpose of iteratively changing the weights of a model to decrease the loss from the loss function[25]. For example, in an image classification problem using supervised learning, the loss function quantifies the discrepancy between the predictions made by the model and the ground truth values. Finding the model parameters that decrease this discrepancy as much as possible becomes the aim of the algorithm, enabling the model to learn from the data. As such leveraging the loss function's gradient, or the first derivative is an integral part of the process. The purpose of the gradient descent is to provide the direction of the steepest ascent, meaning the direction where the loss output increases the fastest, then take a step in the opposite direction. The goal is, therefore, to take a step in the steepest descent incrementally until the minimum is reached. When a minimum is eventually reached there is still no guarantee that the optimal model has been achieved as it is uncertain whether the minimum has been met is the global or a local minimum. Stochastic gradient descent is the most basic form of this gradient descent but other functions like RMSProp, which introduces an adaptive learning rate, and ADAM which is Momentum in addition to adaptive learning, have been developed to explore better means of finding the global minimum. These are presented below.

2.3.1 Stochastic Gradient Descent

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla f(\mathbf{w}_t) \tag{5}$$

Stochastic gradient descent (SGD) has emerged as a popular optimization technique for training machine learning models, deep neural networks in particular. It is high in both efficiency and robustness in dealing with large-scale datasets, which are common in modern ML applications. It differs from traditional gradient descent, where the entire data set is processed to compute the gradient of the objective function, SGD approximates the true gradient by considering a random subset, or mini-batch, of the data at each iteration. This randomization introduces inherent noise into the optimization process, reducing the computational burden and enabling SGD to escape shallow local minima and saddle points that may hinder the convergence of other optimization methods. Furthermore, SGD can be easily combined with momentum-based techniques, adaptive learning rates, and regularization strategies to improve its convergence properties and generalization performance. Despite its simplicity, SGD has proven to be a powerful tool for solving complex learning problems, contributing significantly to the advancement of state-of-the-art machine learning algorithms and applications.

2.3.2 RMSprop

$$\begin{aligned} E[g^2]_t &= \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \\ w_{t+1} &= w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{6}$$

RMSprop, short for Root Mean Square Propagation, is an adaptive optimization algorithm that has gained significant attention in the field of machine learning, particularly in training deep neural networks. Introduced by Geoffrey Hinton, RMSprop is designed to address the challenges posed by the non-convex optimization landscapes that are typical in deep learning problems, such as vanishing or exploding gradients and poor conditioning. The central idea behind RMSprop is to maintain a running average of the squared gradients for each model parameter and to normalize the parameter updates with the square root of these averages. This normalization effectively adapts the learning rate for each parameter, allowing the algorithm to make larger updates for infrequently updated parameters and smaller updates for those that change more frequently. As a result, RMSprop exhibits a more stable and efficient convergence behavior, particularly in high-dimensional, non-convex optimization problems. Moreover, the algorithm is relatively simple to implement and has only a few hyperparameters, making it easy to incorporate into existing optimization frameworks. The widespread adoption of RMSprop in training deep learning models has facilitated the development of more accurate and robust algorithms, which has in turn led to substantial advancements in various applications, including computer vision, natural language processing, and reinforcement learning.

2.3.3 ADAM

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
\tag{7}
$$

The ADAM (Adaptive Moment Estimation) optimizer, proposed by Kingma and Ba, has rapidly become a popular choice for training machine learning models, particularly deep neural networks, due to its remarkable effectiveness and efficiency in handling complex optimization problems. As an adaptive optimization algorithm, ADAM combines the advantages of two key techniques: momentum-based methods and adaptive learning rates, which are exemplified by RMSprop. By maintaining separate running averages of both the first moment (mean) and the second moment (uncentered variance) of the gradients, ADAM dynamically adjusts the learning rate for each model parameter throughout the optimization process. This adaptive behavior enables the algorithm to navigate high-dimensional, non-convex landscapes with greater ease and precision, often converging faster and more consistently than other optimization methods. Furthermore, ADAM is robust to various gradient-related issues, such as sparse gradients, that can impede the training of deep learning models. The algorithm's simplicity and minimal computational overhead make it an attractive choice for a wide range of applications, from computer vision and natural language processing to reinforcement learning and generative modeling. The success of ADAM in both research and industry settings highlights its considerable impact on the development of state-of-the-art machine learning techniques and underscores the importance of continued advancements in optimization algorithms for the future of artificial intelligence.

2.4 Microcontrollers

The battery is as with any device an issue that persists with technological advancement, and with a greater workload placed on the microcontrollers, it is a special issue in need of being addressed. A coin battery commonly used for low-power IoT devices like the Sparkfun Edge 2 and the Arduino Nano 33 IoT is CR2032, it might hold 2,500 J which would last the device roughly a month if running at 1 mW[29]. Since it is desirable to develop something functional on several different devices and thus the potential difference in battery size, making sure that the neural network is able to fit directly on SRAM is crucial for ensuring QoS with heterogeneous technology due to the difference in power requirements based on ease of access to data as discussed in section 2.4.1. As power consumption is an inconvenient bottleneck for the deployment of wireless or low battery size projects, the development of on-device functionality needs to focus on memory usage

and computational costs with a focus on optimising when possible. Battery issues can also come in the form of communication costs however as mentioned in Section 2.6.2, FL might provide regular updates.

2.4.1 Limitations

Memory The difficulties of memory on embedded devices in terms of read-only storage and SRAM does not only concern the ability to deploy the model on the device, but also the capacity at which the device is able to run inference on the model. For optimising battery usage it is desirable for the network to fit onto the SRAM as 32bit SRAM access requires 5pJ, while dynamic RAM (DRAM) requires 640pJ[10]. Tensorflow Lite Micro has been developed with devices with low SRAM in mind, having been designed to work on devices with as low as 4 KB to 20 KB SRAM[29], but the design of the application decide footprint and engineering decisions can have significant influence. Memory optimisation consequently becomes a core principle in lowering battery usage, but will also ensure less computational cost possibly resulting in faster on-device training.

Computational power Like memory and battery, computational capabilities might vary which affects the speed in particular of training. While development on-device must be done with all attributes in consideration for the best possible QoS, the FL algorithm might be sub-optimal with devices with particularly low computational power as such stragglers stifle model updates. Some specific devices or devices in special circumstances might disproportionately affect the speed of the model generation or updating as FedAvg attempts to provide a guarantee of opimisation by synchronously training a joint model[18], meaning that in a large network of devices, specific subsets will have more significant effects. Computational power is therefore something to take into account in the implementation of the FL aspect since different approaches to dealing with slower devices yield different benefits. Abandoning stragglers after a certain timestamp can be an option, or to have specifically dedicated devices responsible for learning whitelisted, or finally to select a random subset responsible for model updates instead of all in hopes of minimizing chances of being affected by stragglers. Accuracy would then need to be compared to time and energy cost saved, and privacy might have to be addressed if some devices are too prevalent in the training subset potentially causing unintentional memorization to develop as described in 2.6.2. Computational power, therefore, needs to be addressed when developing the on-device functionality and taken into account in the overall system.

2.4.2 Optimizations

When developing computationally complex tasks on microcontrollers there are several possible ways of optimizing processes which might rarely be considered in systems with fewer constraints. Independently from strategies tailored for ML model compression, which are more thoroughy discussed in section 2.5, some strategies exist for general arithmetic which serves as a foundation for machine learning optimizations, but are also powerful when

used independently. These strategies include optimizing for efficient memory use, employing fixed-point arithmetic, using function inlining,

Efficient memory usage As mentioned previously and described further in section 3.3, breaching the SRAM of the device and relying on the flash memory significantly increases the power consumption while slowing down computation[10]. Various methods for having efficient memory usage are therefore necessary, most of which encompass good code practice to not be wasteful of resources, while architecture-specific optimization might also be available. Using appropriate data types can reduce memory requirements significantly as 32-bit floats might be sufficient rather than 64-but floats, in addition, limiting global variables can be smart as they can tie up memory unnecessarily while local gets freed when out of scope. Using the 'PROGMEM' keyword on AVR architecture for read-only variables can be smart to place less frequently used variables in flash memory instead of in SRAM so that there is more free space for more frequently used variables, but since it does increase access time not all constant variables benefit from it necessarily. Minimizing the dynamic memory allocation or reusing existing buffers naturally mitigates excessive memory usage as performing operations without intermediary new dynamically allocated variables removes unnecessary initialization and computation steps, while similarly, not leveraging sufficiently sized existing also requires initialization and thus takes up the memory until deallocation takes place. In addition, when calling functions and passing by value, the passed data is copied, thus taking up twice the space necessary in many cases, though can be mitigated by passing by reference when appropriate.

Fixed-point and Integer arithmetic While microcontrollers tend to have a floating point processing unit available, for example Arduino nano 33 BLE through its ARM Cortex-M4 CPU, generally using integer arithmetic is faster. Fixed point arithmetic can be beneficial when there is no need for high-precision arithmetic as it can be faster and require less power than floating point arithmetic. It can be used to make the computation of floats more efficient by instead representing them as integers while dedicating a number of bits in the integer as the decimals, for example by having an 8-bit integer with the 4 left bits signifying integers and the 4 right bits representing the decimals of the original floats. The calculation of the float based on designated bits is shown in equation 8, where for example the binary point number '0011.1100' is the same as '3.75'.

$$D = \sum_{i=1}^{n} b_i \times 2^{-i} \tag{8}$$

The issue with such arithmetic however is that the range and precision is very limited compared to using floating-point numbers and are determined by the bits allocated, making them less appropriate for applications where high precision or large dynamic ranges are required.

Instead of allowing for representing floating-point numbers at all in integers, the numbers can be fully turned into integers which makes the computation more straightforward, while benefiting from the gains in computa-

tional performance. Examples of achieving this integer arithmetic include just multiplying the number by a scaling factor to make the n-th decimal the first digit and then doing the arithmetic on the integer before dividing and having to obtain the floating-point number. In additon, quantization, which is further discussed in section 2.5.2, is a means to map a range of values to a discrete set of values, enabling the developer to map floats to integers. This is commonly used for inference on ML models as it both reduces size and enables leveraging of the faster integer arithmetic on microcontrollers.

Function Inlining When calling functions there is a certain amount of overhead required due to the need to jump to the function's code, push the parameters to the stack, pop the return address off the stack, and finally restore the state when the function returns, these overhead elements are all eliminated by inlining the function. The inlining instead replaces the function with the function body itself, which also allows for better optimization of the code by the compiler. It can also benefit the program by improving the cache locality as the code that is executed is kept together in the same place, which reduces instruction cache misses. The drawback of using inlining however is that it might introduce bloat as it increases the size of the executable due to the function call being replaced with its code. As such, if the function is too large or used a significant enough amount of times, the inlining might be detrimental to the overall performance.

Loop Unrolling In neural networks, there is a lot of calculation using tensors, implementing the operation, therefore, requires a significant amount of nested loops as arrays of different dimensions need to have their individual values accessed. Looping, however, comes with some overhead due to the instructions determining whether to stop or continue, this includes initialization when setting up the loop counter, condition checking to determine if it should continue, counter update, and branching when loop condition is met. A means to remove this overhead is loop unrolling which consists of duplicating code to reduce the number of loop iterations[1]. In an example where a program loops over a function 4 times, the code can either be copied 4 times with relevant indices declared manually, or have the loop the operation twice on index and index i+1, making it only have to loop 2 times instead. Having the loop body executed multiple times per iteration, therefore, negates the potentially unnecessary control instruction, however, it comes at a cost. Loop unrolling means duplicating the code leading to more memory consumption at the cost of potentially minor efficiency gains. Therefore when considering loop unrolling, it is wise to not employ it where the code body to duplicate is large, especially as a larger code again can lead to an increased number of cache misses as well as the memory consumption rising.

2.4.3 Arduino

The Arduino platform has become central in microcontroller-based hardware development with its open-source nature, allowing a multitude of cre-

ative, educational, and professional innovations through its ease of development. Its design, characterized by a physical board hosting a microcontroller, is complemented by a set of input and output pins that provide an interface for an array of sensors, actuators, as well as other peripheral devices, making it a versatile choice for various applications. The platform's strength also lies in its integrated development environment (IDE) due to its user-friendliness. This allows developers to write code in a high-level, simplified C++ language, upload it to the Arduino device, and execute the program. The IDE is versatile in that it promotes accessibility, makingf Arduino serve as a suitable starting point for those exploring embedded systems development, while also offering enough depth for seasoned developers to build complex systems.

The Arduino Nano 33 BLE, a variant of the Arduino platform, covers all of these features in a more compact form while adding support for Bluetooth Low Energy (BLE) communication. The BLE capability expands the range of possible applications, particularly ones related to the development of IoT, where the need for low-power, short-range wireless communication is especially important, commonly with an edge device. Furthermore, the Arduino Nano 33 BLE contains the nRF52840 microcontroller from Nordic Semiconductor. It is very power-efficient, while boasting a 64 MHz clock speed, 1MB of flash memory, as well as 256KB of SRAM, offering a significant leap in computational capabilities for IoT applications. It also provides an FPU and digital signal processing instructions, making it particularly suitable for running complex algorithms such as those used in machine learning.

2.4.4 Sparkfun

In addition to Arduino, SparkFun has become a prominent participant in open-source hardware contributing to the democratization of the development of electronics and embedded systems. They offer a broad range of products, from sensors and microcontroller boards to robotics and machine learning kits. One of SparkFun's notable development boards in the realm of microcontrollers and machine learning is the SparkFun Edge Development Board, which is powered by the Apollo3 Blue microcontroller from Ambiq. The Apollo3 Blue comes with 1MB flash like the arduino, but provides 384KB SRAM. It is especially significant in this realm as it was developed with power efficiency in mind, making it particularly suitable for edge computing and ML applications where power consumption is such a critical consideration. The Apollo3 Blue microcontroller integrates a high-performance ARM Cortex-M4F processor, which by default runs at 48MHz, but is capable of 96MHz with its burst mode, while also offering a low-power mode that consumes less than 5µA/MHz. This combination of high-performance processing and low-power operation has made the SparkFun Edge a solid platform for developing power-sensitive IoT and ML applications.

2.5 Model Compression

As the constraints of microcontrollers have been highlighted, and optimizations shown previously for general arithmetic and code execution, specific

methods to compress and therefore optimize memory consumption and speed of ML models remain as a core aspect for on-device deployment. Compression techniques attempt to reduce inference speed or number of weights, or the size of existing weights through various means and for various purposes. For model size compression, knowledge-distillation can be used as it created a high-performing smaller model based on a more complex, while for inference only, weight quantization can be used as decreases the memory requirement of each weight for example from 32-bit float to 8-bit integer. While all model size compression optimizes for speed for due to requiring less computation, other specific optimizations exist, for example, pruning which removes unnecessary connections, allowing for faster calculations, and in addition, weight quantization is worth to be mentioned as it increases inference speed by allowing integer arithmetic as described in section 2.4.2. Finally, weight clustering (Huffman-encoding) is a means to compress the weights to allow for faster transmission of weights between devices, as they would have to be decompressed before use in inference or training. All of these compression methods will be more thoroughly described below.

2.5.1 Weight Pruning

Weight pruning is a popular model compression technique used in deep learning that aims to reduce the number of parameters in a model without significantly sacrificing accuracy. With the ever-increasing complexity of deep learning models, model compression techniques have become essential for enabling the deployment of deep learning models on resource-constrained devices. It works by selectively removing or ”pruning” weights in the network that contribute less to the overall performance of the model. The idea is to remove redundant connections and parameters from the network, resulting in a more compact and computationally efficient model. By removing unnecessary weights from the network, weight pruning reduces the model's memory footprint and the number of floating-point operations required during inference, resulting in faster and more energy-efficient models.

2.5.2 Quantization

Quantization is a popular technique used in model compression that aims to reduce the storage and computational requirements of deep learning models. Deep learning models often contain millions of parameters, which can make them challenging to deploy on resource-constrained devices such as smartphones and embedded systems. Model compression techniques such as quantization are therefore essential for enabling the deployment of deep learning models in such environments.

Quantization involves representing the parameters of a deep learning model using fewer bits than their original representation. The most common type of quantization is integer quantization, which represents the model's parameters using integers instead of floating-point numbers. The number of bits used to represent each parameter determines the precision of the quantization. For example, if each parameter is represented using

8 bits, it is called 8-bit quantization. Quantization can be performed in several ways, including post-training quantization and quantization-aware training. In post-training quantization, the model is first trained using floating-point arithmetic and then quantized to an integer representation. Quantization-aware training involves training the model directly in its quantized form, ensuring that the quantized model's performance is similar to the original model. While quantization can significantly reduce the storage and computational requirements of deep learning models, it can also lead to degraded model performance if not done correctly. To ensure that the quantized model's accuracy is not compromised, the quantization process must be carefully tuned to balance the trade-off between model size and performance. This can involve selecting an appropriate quantization bit width, selecting a suitable quantization algorithm, and fine-tuning the quantized model to ensure that its performance is similar to the original model. Quantization can be used in combination with other model compression techniques, such as weight pruning, to further reduce the model's size and improve its efficiency. While quantization can be challenging to implement, it is an effective technique for compressing deep learning models, making them more efficient and enabling their deployment on resource-constrained devices. As such, it remains an active area of research in deep learning and model compression. New and improved quantization techniques and algorithms are being developed to achieve better compression rates and model performance.

2.5.3 Knowledge-Distillation

When desiring to deploy a smaller version of a model to more a resource-constrained device, a means to extract the knowledge from the bigger model into a smaller can be beneficial to mitigate having to re-train, while potentially achieving better results than though such re-training. Knowledge-distillation provides this by having a larger and more complex model denoted as the ”Teacher” model transfer its knowledge to a smaller and simpler model denoted as the ”student”[4]. By doing this, the goal is to leverage the power of the resource-intensive complex model on a smaller model while maintaining efficiency. The knowledge distillation process takes place by first of all having the student replicate the output of the teacher, but more significantly mimic the entirety of the behaviour of the teacher in general. This means having the student learn the representations the teacher learns and how it generalizes, which is done by having the output probabilities of the teacher serve as soft targets, not hard ground truths, as they provide richer and more informative training signals due to them capturing the teacher's certainty of its predictions. When training on such probability distribution the student can implicitly learn how the teacher might miss categorise slightly and provide higher probability output to incorrect classes. After this training process, the student should have comparable performance to the teacher.

2.6 Training Paradigms

2.6.1 Centralized Learning

As science and technology surrounding AI and microcontrollers developed independently, the combination of the two focused on leveraging the individual technologies' strengths, through the traditional means of having a centralized AI server with external microcontrollers. While it garnered beneficial results, the architecture of such solutions comes with inherent flaws whose solution requires a change in the fundamental structure of the service. The issues concern problems surrounding privacy, training and energy cost, latency, and data-transmission volume[2].

Privacy Data privacy is an emerging issue as a regulation requiring strict procedures for preventing unwanted access to private data emerges. As such, privacy becomes both an ethical and legal aspect that has to be addressed for any products. The issues come in the form of training data sets used for the machine learning model, in regards to how the data is stored in a dedicated facility which might be vulnerable to leaks as they become a target for attacks. It opens up more security holes in terms of personnel working there as well since if given access, the security of the data depends on the individuals' integrity. In addition, edge-device sensors providing the data to the model, either for processing or perhaps for training could inherently have the flaw of directly providing potentially sensitive data which may be accessed by others[27]. In the healthcare sector, these could be measurements made by small sensors which together might disclose private information about the individual. The presence of such issues makes the service go against the General Data Protection Regulation[5] (GDPR) as well as other national and international data privacy laws, which would make individuals deploying the technology liable for fines while stopping the operation of the service.
Communication between devices and servers as well as the training process can be restricted from posing privacy risks through decentralization of the intelligence by bringing more of the processes on the edge thems could enable training to take place by having each edge device train locally and thereafter pass the weights to a center which aggregates them to a model based on the distributed training[18]. Thereafter, the edge devices would no longer need to communicate with central intelligence for inference as it can be handled locally, removing the privacy vulnerability of transmission of data.

Cost: Training, electricity, and CO2 Traditional centralised learning comes with high cost in terms of time, electricity, and potentially CO2 production as it relies on being able to train potentially multiple networks to a sufficient standard for deployment. This is especially an issue with the issue of a growing number of heterogeneous devices needing inference functionality, meaning that similar networks might be trained for a number of times consistent with their number of deployment scenarios[3] increasing the cost as a result. Figure 5 illustrate different centralised learning models with their training time, CO2 emissions, and cost of train-

ing on Amazon Web Services if the number of deployment scenarios is 40. The figure makes it clear that requiring regular training of new models is both costly and comes with some CO2 emission consequence which has fortunately been reduced over time. Continuously generating new models in this manner quickly adds up time, however, especially given that the amount of IoT devices in use is increasing rapidly as it was estimated that 250 billion microcontrollers were deployed in 2020 and that it would increase by 40 billion in 2 years[27]

Reducing Latency Devices communicating with one another will require some varying degree of latency which affects the efficiency of the service provided. As a minimum of two messages needs to be sent if inference happens on the server, the reaction from sensing to acting for microcontrollers cannot be assumed to be close to instant or close to average inference time, as communication time, as well as inference time, can be varying depending on connection and server availability. Taking this into account, the application of such solutions will be restricted in environments where safety outweighs the need for autonomy, potentially missing large markets and overall indicating an undesired lack in quality which any firm would benefit from not being present. Therefore, reducing or even eliminating latency is paramount in ensuring the viability of this technology as well as widespread adoption. An intuitive approach to fixing this is decentralising the inference to the edge devices themselves to eliminate the communication time and perhaps gain more consistent inference time as is discussed further in later sections.

Re-usability The centralized ML solutions may inhibit portability and thereby re-usability due to the nature of privacy concerns in how the intelligence is trained, as well as architecture not being accommodated for change in the environment without potentially expensive updates in later iterations. Ideally an architecture better suited for varying environments where several topic-specific models can be deployed which potentially learn based on the individual situation, not based on a fully predefined model. An example of this is how Internet Service Providers (ISPs) might desire to classify the applications of the packets with encrypted payload flowing through the network for security and quality of service purposes as presented in the following paper [21]. In such an example centralized machine learning would be detrimental to the longevity of the model as the emergence of new applications would render it less usable over time unless the new data is manually gathered for updating the model. Simultaneously training such a model comes with significant privacy concerns as training depends on data of user traffic which is among what the GDPR aims to protect. Re-usability concerns itself with the portability of the solution in that the model and models with no or minimal degree of adaptability which needs large iterative updates are not necessarily reusable outside their exact context and potentially not outside a specific time frame. Architecture that is able to provide a greater variation in data received, preferably with built-in data-sampling techniques for enabling real-time model updating is a potential solution that can improve existing services that deal with chang-

ing environments (e.g. ISP traffic classification, or industrial contexts in which technological components are replaced and improved). A centralized solution for this is online learning which enables continuous model training with the introduction of new data, and thereafter adjusts the model with the potential of converging equally fast as batch learning[16], however, it likely still suffers mentioned centralised learning pitfalls. This would also address re-usability in other contexts as generic models and software can be provided and improved based on specific surroundings, instead of building new context-specific models from centralised data set with each new ML solution. In the example of traffic classification, providing access to local data on external devices for training can assist in improving performance over time and discovering the new emerging applications over time[21], consequently enabling both model and implementation re-usability to a greater extent than traditional implementations. The motivation for this is saving cost in terms of money and electricity for updates, which in turn concerns itself with the environmental impact of the solution, an increasingly important aspect of ML and artificial intelligence in general as carbon emissions have become a factor to consider[6]. Carbon emission is significant independently of centralised learning, but there are indications that alternative ML solutions like Federated Learning show lower emission[22], which is achieved through for example negating the need for massive cooling systems. However, variables such as device location, the primary energy source in the country, model architecture, and aggregation strategy, can affect this significantly for better or worse.

2.6.2 Federated Training

Federated learning (FL) is a machine learning technique developed by Google with the intention to enable independent mobile devices to train ML models collaboratively without sharing raw data, thereby decoupling ML training from a central cloud data storage[2]. Its primary intended area of application was initially mobile phones, due to its potential to improve existing AI-assisted processes to eventually improve user experience, these include next-word-prediction, text entry on touch-screen keyboard, and voice recognition [18].

Sensitive Data A primary motivation for the implementation of FL in favour of data center training on persistent data is the inherent qualities ensuring the protection of the data itself as well as the identity of participating devices[18]. Privacy for sensitive data is enhanced through the architecture since it enables differential privacy (DP), which is a means to describe the patterns in a data set without disclosing the individual contents, meaning that a central server has no access to raw data[2]. As its initial intended purposes were for mobile phones, FL was also developed with anonymisation in mind as it was undesirable to be able to trace back to the device based on updates received in case there is any parse-able sensitive data within the weight updates[2]. In order to respect the principle of data minimisation, three particular principles are incorporated into the structure. These are focused collection, i.e. only transmitting minimal updates and limiting local data access at all stages, early aggregation, i.e.

processing individual devices' data as early as possible, and minimal retention, i.e. discarding processed and collected data as soon as it can[2]. The area is however continually developing since, after its inception, further development focused on improving privacy has emerged. DP-FedAvg was for example developed with the intention of improving device level DP which helps ensure that training is not too sensitive to one or more specific devices over others[2]. In addition, algorithms for empirical privacy auditing have been developed for discovering unintended memorisation in FL potentially caused by unique sequences from specific devices when training to further optimise anonymity[2].

Algorithm 1 Federated Averaging Algorithm. (FedAvg) [18]
clients with index k, E is number of local epochs, B is local minibatch size, and η is learning rate

---

Server Side

initialize w_0
for each round $t = 1, 2, ...$ **do**
  $m \leftarrow max(C \cdot K, 1)$
  $S_t \leftarrow$(random set of m clients)
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow ClientUpdate(k_t, w_t)$
  end for
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w)**:**

$B \leftarrow$ (split P_k into batches of size B)
for each local epoch i from 1 to E **do**
  for batch $b \in B$ **do**
    $w \leftarrow w - \eta \Delta \ell(w; b)$
  end for
end forreturn w to server

Federated Averaging Algorithm Federated learning was first introduced using the federated averaging algorithm as described in algorithm 1. It works by training for t number of times, by taking a fraction C of the total number of clients K and selecting a random set of m clients. For each client k, the current rounds weights w_t are sent to the client which runs stochastic gradient descent for E epochs and then returns updated weights to the server. The server thereafter aggregates the weights to obtain the average weight which is to be used in the next iteration[18]. In order to obtain the best-shared model across clients, the federated averaging algorithm uses a global loss which is a weighted average of each client's loss as shown in equation 9. For each client k it computes the loss on the client-side using its loss function $F_k(w)$, commonly cross-entropy loss, which is thereafter weighed by the size of the clients' data set. The devices with larger data set size will thereby have correspondingly larger losses and weights[18].

$$f(x) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \tag{9}$$

Model Size An issue present both in the case of models generated by centralized learning as well as those of FL is the model size. This issue is of particular concern for embedded devices as more than 1 MB of available storage is rarely found, and static RAM (SRAM) is often either 512 KB or less[29]. Size increase in models makes memory a bottleneck for deploying them for inference, especially in the context of microcontrollers as the performance gains from increasingly improving models cannot fit into memory. The model growth over can be seen in figure 3.

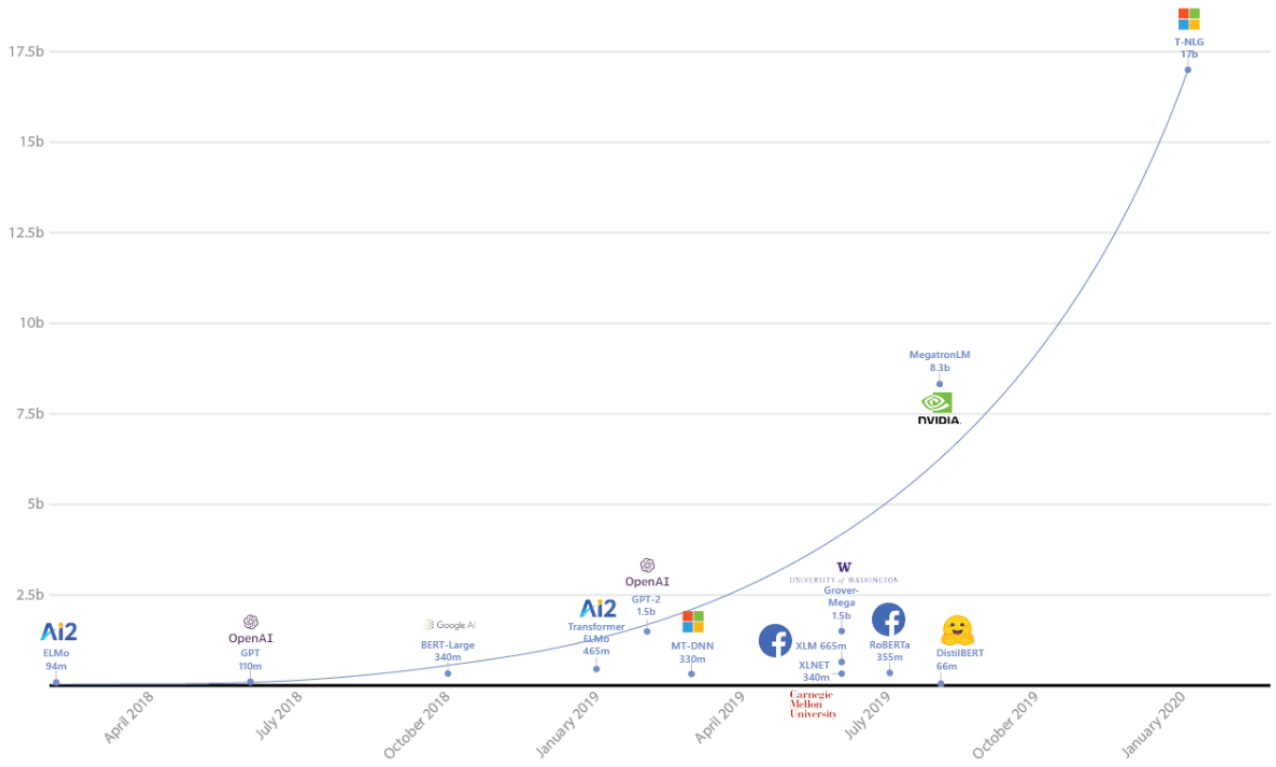


Figure 3: Growth of model size over the years[19]

Communication cost Traditional solutions suffer power costs surrounding the abundance of potential communication, where microcontrollers might have to send raw data to a central server for inference and thereafter receive a decision. FL solves this by enabling the devices to run the inference themselves, saving the communication which was required with each classification, however, communication cost will now come in the form of passing of weights to other devices or receiving updated weights[18]. This

communication is commonly done by having a common fog node request model updates, then receiving them to perform the federated averaging algorithm as seen in algorithm 1 and thereafter pass updater weights back. Whether this approach induces more power cost depends on the specific implementation, but generally, a significant reduction is to be expected, as the developers of the federated averaging algorithm reached a 10-100X reduction in required communication rounds compared to synchronized stochastic gradient descent[18].

2.7 Evaluation Metrics

2.7.1 Classification Accuracy

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{10}$$

Classification accuracy is a commonly used metric for evaluating the performance of machine learning models in classification tasks. It measures the percentage of correctly classified samples out of the total number of samples in the test set.

2.7.2 Mean Absolute Error

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{11}$$

Mean absolute error (MAE) is a popular loss function and metric used in machine learning for regression tasks. It measures the average absolute difference between the predicted and actual values of a target variable. Unlike mean square error (MSE), which penalizes large errors more than small errors due to the squared term, MAE treats all errors equally. It has several advantages over MSE. Firstly, it is more robust to outliers since it does not heavily penalize large errors like MSE does. Secondly, it is easier to interpret since it is in the same units as the target variable. Finally, it is computationally more efficient since it does not involve the costly squared term. MAE can be used as a loss function in various machine learning algorithms, including linear regression, decision trees, and neural networks. Like MSE, MAE is a differentiable and continuous loss function that can be optimized using gradient-based methods, such as stochastic gradient descent.

2.7.3 Mean Square Error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{12}$$

Mean square error (MSE) is a popular loss function used in regression tasks that measures the average squared difference between the predicted and actual values of a target variable. However, it is not commonly used in classification tasks since the output of a classifier is usually a categorical

variable, rather than a continuous variable. In classification tasks, cross-entropy loss is the most commonly used loss function.

2.8 Mean Absolute Percentage Error

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \tag{13}$$

The Mean Absolute Percentage Error (MAPE) is a core metric in the field of predictive analytics and forecast accuracy and works by quantifying the level of error in predictions involving continuous variables. The metric, as seen in equation 13 is calculated as the average of the absolute percentage differences between ground truth and prediction made by the model. It offers the advantage of scale independence, which makes it useful for comparing the accuracy of different models and data sets. A drawback of the metric however is that in scenarios in which the ground truth holds zero or near-zero values, the computation of percentage error could lead to extremely high which distorts the overall error metric. As such

Also, as it is an absolute metric it does not differentiate between over-prediction and underprediction, which might be necessary

2.9 Edge ML Architectures

2.9.1 Edge vs cloud

As the computational power at the edge increases and concerns regarding privacy and latency issues with relying heavily on a distant cloud increases, edge computing as a computational paradigm has gained prominence. It seeks to bring computation and data storage closer to the location where it's needed, with the goal of achieving better response times and save bandwidth, as well as reducing the privacy concerns with transmission and storage on the cloud. The desire to have local processing and storage is particularly beneficial in the context of the emerging IoT systems, as vast quantities of data are produced by an immense number of devices, which would be time-consuming and expensive to transmitt to a distant server for processing. Edge computing offers several significant advantages compared to traditional centralized computing. First and foremost, by processing data closer to users or IoT nodes it can significantly reduce latency, as near real-time responsiveness can be achieved, which is crucial for time-critical applications such as autonomous vehicles, industrial automation, and real-time data analytics. Second, edge computing can save on bandwidth and reduce network congestion by reducing the need for long-distance data transmission, which leading to more efficient use of network resources. lastly, processing data locally can enhance the privacy and security of the data, as sensitive data doesn't have to leave the local network making transmission and storage far less of a concern. This is especially because there would no longer be a single point of failure which could compromise all data, in the event that an edge gets compromised, the effect is far less significant.

There are certain challenges posed by edge compuring however as they tend to have significantly less computational power and storage capacity.

This limits the complexity of potential tasks and deployment scenarios compared to the cloud counterparts. In addition, the management of edge devices, especially when considering connected IoT devices will be significantly more complex than simply managing a cloud. The advancement in both hardware and software however continues to demonstrate the potential of edge computing as techniques for model compression and hardware acceleration are created and allow for new use-cases, as well as techniques like federated learning which makes significant use of edge nodes for a complex task without as high requirements as traditional centralized learning, as such, edge-computing has rapidly emerging new efficient, responsive and privacy-preserving applications which are competitive with similar cloud solutions.

2.10 Implementation Technologies

2.10.1 TensorFlow

Tensorflow is a software library for numerical computation and machine learning commonly used in both industry and academia to develop and deploy ML models. It is particularly used for deep learning, such as image recognition, speech recognition and language processing. Tensorflow supports a range of languages, including Python, C++ and Java, and in the case of Python provides a high-level API called Keras which simplifies the process of building and training further increasing its popularity. Because of this abundance of functionality which needs to be heavily optimized for performance improvements, it it typically used on desktops and servers.

2.10.2 TensorFlow Lite

Because of the vast array of functionality and complexity in optimization, standard TensorFlow is however unsuitable for smaller devices like phones. Due to the limitation in resources, deployed models on phones is likely to have slower performance and therefore increased energy consumption. While this is caused by a model and library size issue, it can also be attributed to differences in hardware and software architecture. Mobile devices tend to use different processors and operating systems, making architecture a barrier to optimizations, and making the quality of developed models suffer further.

To address this issue TensorFlow Lite has been developed to fit on mobiles with mobile-specific optimizations. It includes a subset of TensorFlow functionality suitable for mobile applications, and tools for model optimization and compression for improved performance.

2.10.3 TensorFlow Lite Micro

TensorFlow Lite Micro is a subset of TensorFlow Lite designed for more resource-constrained devices like microcontrollers, and is therefore optimized for small-memory, low-power devices such as wearables, sensors, and other IoT devices. It allows for running models on devices with as little as 16kb as the subset of operations available is small and heavily optimized.

Because of the resource constraints, it has however been developed with inference in mind as such it lacks on-device training capability in favor of more operations for just inference and a smaller size to fit more devices. Backpropagation requires space to store each weight's gradients and learning rate, roughly doubling the memory requirement during training, while optimizers like ADAM can cause the memory consumption to triple. As such, adding training capability for all operations would increase library size significantly, and having the capability at all is low priority due to performance issues.

3 Related work

As the application of ML techniques on resource-constrained devices has gained significant attention in recent years with the emergence of IoT in the form of wearable gadgets, mobile phones, and devices for industrial automation, researchers have explored various strategies for implementing tiny ML. To efficiently implement sufficiently accurate, fast and cost-effective ML models on such constrained devices core areas studied include, but are not limited to, on-device training, federated learning, and model compression. As such, this related work section aims to provide an overview literature considered the most pertinent in this area, highlighting challenges faced in this area, the methodologies, and key advances achieved by the researchers. The review is structured in three themes: (1) ML training on microcontrollers, focusing on techniques and methods for training ML models with respect to resource constraints; (2) Federated learning, which explores the advances in the decentralized learning paradigm to enable collaborative model training using several devices to address privacy and cost concerns, the paradigm is explored independently and in relation to microcontrollers, finally (3) model compression is investigated to understands means of reducing model complexity and size to enable deployment on resource-constrained devices. By examingint he developments in these areas, the goal is to comprehensively cover the current state of the art and potential avenues for future research for ML on microcontrollers.

3.1 Current standing of on-device training

While ML model inference for resource-constrained devices has been extensively researched and tested, resulting in libraries such as Tensorflow Lite & Lite Micro and PyTorch Lite, actual on-device training has garnered less attention. This section is dedicated to the discussion of notable works and implementations which cover this area. The works include TinyFedTL by Kavya Kopparapu and Eric Lin[15], the on-device training work by Grau et al.[17], development of tiny Online ML by Haoyu et al.[24], Ravaglia Et al.'s platform for tinyML continual learning[23], Over-the-air tinyML model deployment by Sudharsan Et al.[28], and finally EtinyNet by Xu Et al.[7]. Collectively these works aim to advance the state-of-the-art of ML model training on IoT and contribute to ultimately facilitating viable implementation with and without FL.

3.1.1 On-device training of FC-layers

TinyFedTL is a notable implementation of federated learning on microcontrollers done by demonstrated by Kavya Kopparapu and Eric Lin[15]. The primary objective of the implementation is to enable on-device model training for the purpose of federated learning to allow continuous learning for improving mode over time using locally stored data. A significant challenge faced is the aforementioned issue of known tinyML frameworks lacking support for such on-device training in favor of only supporting inference on static models. To address the challenge, the resulting implementation

thereby demonstrates on-device training by removing the final fully connected (FC) layer of a CNN to create a feature extractor. The CNN is converted into a Tensorflow Lite Micro model which is deployed on the microcontroller and used for inference through the interpreter, however without any training capability. This is followed by a self-implemented FC layer, which enables the conversion of features gained from CNN to be parsed into probabilities through FC layer and a Softmax activation function. Lastly, a self-implemented backpropagation function allows training of this final FC layer for the actual on-device training to take place. The need to self-implement these ML components rather than relying on existing tinyML libraries demonstrates a lack of standardization and state-of-the-art solution for training on resource-restricted devices, further showing an inhibiting factor in the general research and adoption of federated learning in the context of IoT. As such, the need to reinvent the wheel by creating these components is a consistent issue in all related research.

The TinyFedTL[15] effectively demonstrates the viability of on-device training in both accuracy and speed. It makes use of ImageNet and Visual Wake Words dataset and two compressed versions of MobileNetV2 models taking up 210kB of DRAM and 657KB of program storage but does not face any further memory demands due to FL allowing input data to be discarded and not stored post model update. The findings indicate that number of devices affects validation accuracy negatively, and a proposed cause is that individual progress can be cancelled out by the weight averaging of the FL algorithm. The accuracy also showed to stabilise around 3000 training examples, indicating issues of reaching local optima potentially due to the simplicity of optimiser used. In terms of performance, the TinyFedML implementation spent 8-10 seconds on the image capturing and inference due to having buffer from Arducam data processing, training 20 episodes of local epochs took 214ms, and upload/download of weights and bias data took over 30 seconds each way. As such it demonstrates that there is gain to be had in pursuing further advancement in the technology, in particular regarding function for optimization and support for better weight decay, FC layer implementation and communication cost.

Work seeking to address on-device training issues has emerged following TinyFedML, an example of with is ”On-Device Training of Machine Learning Models on Microcontrollers with Federated Learning”[17] by Grau, Marc Monfort, Roger Pueyo Centelles, and Felix Freitag. The focus of and greatest contrast from TinyFedML[15] is that rather than starting off with a predefined model it aims to generate one from scratch to gain insight into trade-offs from FL design space in the context of the resource constraints present on the microcontrollers, and experimentation is done using keyword spotting as ML purpose with custom created dataset. As the network is not pre-made with a suitable size, a custom-feed forward neural network using a single 25-node hidden layer was defined, with 650 node input layer and a 3 node output layer. The network has a sum of 16325 weights and 28 biases represented by 4B floats making up for a total size of 65412B (63.97kB) which is able to fit on most SRAM. In addition, it seeks to handle gradient descent more optimally and does so using the hyper-parameter momentum to maintain a consistent direction in the descent by combining

the previous vector with a default value of 0.9.

The resulting implementation benefits primarily from the size difference in the network as the computational and communicative cost is reduced[17]. As communication requires a total of 130kB, sending a data sample required 2.49s. The FL algorithm naturally executed at different speeds depending on the number of participating clients; 1 client required 7.12s, two required 7.85s, and three 8.22. The resulting performance of the implementation is expressed in loss epochs with a potentially different frequency of federated learning rounds and explored 4 particular points of interest: efficiency of training with no FL, the effect of the number of FL rounds, hidden layer size effect and effect of training on non-IID data. Individual training on each device showed a decline in loss over time but could have significant spikes in later epochs higher than initial ones indicating inconsistency likely caused by lack of data. With federated learning introduced the decline in loss happens far sooner, and with a FL round at every 2nd epoch, the loss becomes consistently low after 25 epochs, while with a FL round at every 30th epoch same loss result is achieved between 60 and 85 epochs depending on the device. This illustrates the trade-off present in communication cost versus loss, as frequent FL rounds decrease loss more rapidly in terms of the number of epochs but based on execution time and potential power consumption cost it might be faster or more optimal long term with less frequent FL rounds as the communication between devices is reduced significantly which would be increasingly visible with the number of devices. An issue seen in TinyFedML was a decrease in accuracy with the number of devices[15] as mentioned which was likely caused by significant improvements from devices fading out with each weight averaging. As such with a lower frequency of FL rounds, their on-device improvements might become more visible as differences in weights can become significant enough to provide more of an impact, however, this was not addressed. The number of nodes in the hidden layer affected loss over epochs in that a higher amount seemed to be better, the tested amounts were 5, 10, 20, and 25, and likely higher amounts would decrease loss faster. The downside of more nodes is that it would be at the cost of model size which affects the speed of computational inference, training, FL, and communication speed, and can risk not fitting on SRAM. Non-IID-data is quickly trained on each separate device with no FL if trained to a specific word, but when averaging weights the loss is increased significantly with great variation in each epoch, though seemingly declining slowly. As such different aspects of significance in the design of federated tiny ML are explored and vulnerabilities in architecture are made apparent, indicating a need for tried and tested tools specifically for on-device training.

Even though training of ML models on microcontrollers proves to be possible when self-implementing the models in their entirety not relying on inference using static model interpreters, a hybrid of static and trainable models is the more common approach. While TinyFedTL uses a CNN feature extractor before a FC layer, Haoyu et al. deploys an autoencoder which is followed by their self-implemented TinyOL system with trainable weights[24]. The purpose is to enable incremental on-device training on streaming data to allow for model adaptation based on new working conditions and general model improvement over time. As such the core fea-

ture proposed is a manner in which to improve independent devices regardless of each other over time for different circumstances, allowing for expansion to support new classes as they are encountered in the datastream. The TinyOL approach was demonstrated with two anomaly detection autoencoder use cases, namely Fine-Tuning, and Multi-anomaly classification, and the experiment was conducted by collecting vibration data from a USB fan that had three states, either normal, tilted, or stuck. The Fine-tuning approach consists of adapting the pre-trained autoencoder, as the initial model will have a different reconstruction error distribution due to deviations from the training data in the positioning of the microcontroller. When applying the TinyOL system for 2000 iterations using data streamed through Bluetooth, the fine-tuning of model weights is performed as the model adapts to the new rapidly as only minor changes are needed on the pre-trained weights. The resulting reconstruction error distribution is thereafter significantly more similar to the distribution generated by the training dataset.
Multiclass-Anomaly classification differs from fine-tuning by instead having the autoencoder provide a classification at runtime and use it in addition to the construction error as input features to classify status incrementally. The training was thereby conducted by regularly switching the position of the device while providing labels to emulate real-world variability, in addition, if provided with an unknown new label, the TinyOL layer can accommodate it by updating the layer structure for a new class. When testing the performance, the F1 score on the normal class showed to be significantly higher than the two anomaly classes which was speculated to be caused by the autoencoder only having been trained on the normal state and not anomaly. This indicates the need for more layers compatible with training as static models are likely to eventually inhibit the quality. When comparing online and offline training the figure, it is apparent that the amount of data has a significant impact on the model performance, and that the performance of offline training outperforms online after 50 epochs of training.
It can therefore be concluded that if the pre-trained model is sufficiently generalized, the addition of an adaptable layer, whether TinyOL[24], or a simple FC layer as demonstrated in TinyFedTL[15], on top of the static component allows for high performing fine-tuning of models with limited on-device training. The approach of not deploying a static model, but instead a fully trainable model demonstrated by Grau Et al.[17] showed promise as even with a small hidden layer size in the range between 5 to 25 neurons had a sufficient decrease in loss and claims comparable accuracy to server trained model is achieved. However, the approach is simple as it is just a single hidden layer of DNN, and the complexity is high as 80% of the SRAM is consumed primarily due to the weights associated with the 650-node input layer and 25-node hidden layer, as such exploring alternative neural networks approaches for the use-case can be beneficial. In addition, as these approaches consist of only trainable FC layers, need to explore fully trainable neural networks is evident as more advanced approaches often simply rely on the static model.

3.1.2 On-device CNN

A CNN implementation aimed at resource-constrained microcontrollers with SRAM of at most 320KB and 1MB Flash memory has been explored by Xu Et al.[7] resulting in their EtinyNet CNN architecture designed for high parameter efficiency. The necessity for specialized architecture for microcontrollers is made visible by the fact that conventional NN architectures designed for devices with more limited resources, commonly mobile devices, still greatly exceed the capacity of microcontrollers. For example, MobileNetV2 after 8-bit quantization still exceeds the 320kb SRAM limit by 250%[26]. As such EtinyNet proposes an adaptive scale quantization (ASQ) method to alleviate the undesired effects of known quantization schemes while improving the accuracy of quantized tiny models. This is achieved in part by using linear depthwise blocks (LB) and dense linear blocks (DLB). LB is a depthwise convolution followed by a pointwise convolution and finally another depthwise convolution, the resulting block enhances parameter efficiency by increasing the proportion of depthwise convolutions. DLB serves as a dense connection into LB used to increase the width of the network to compensate for the restrictions in the number and size of CNN feature maps. This is done by concatenating the input feature map with the output feature map of the pointwise convolution. The stacking of these LBs and DLBs thus allow for lower memory runtime than when using solely conventional convolutional layers.
The second proposed method, ASQ, addresses the shortcomings of DoReFa quantization which is apparent when used on low-bit widths like 4-bit. It works by rescaling the original weights of the CNN by using a trainable parameter to control new rescaled weights' distribution smoothness, thereafter, clamping the weights to fit the range [-1, 1] is done using the tanh-function, and finally, a linear quantization function is applied. The parameter being trainable allows ASQ to balance minimizing quantization error and maximizing information entropy by adapting to the optimal distribution of quantized weights for each layer. As a result with a model size of 340KB, EtinyNet 0.75 achieved 57% top-1 ImageNet accuracy when using the ASQ quantization, however with DiReFa quantization it achieved only 43.3%. In addition, when tested on a commercial microcontroller it achieved 56.4% on Pascal VOC. EtinyNet therefore especially demonstrates how CNNs can leverage more creative architectural choices as layers and their interconnections can have more variability than traditional DNNs making it evident that these layer connections are especially in need of more research.

3.1.3 Remote model deployment

Whether in the context of federated learning or for ML capability on independent devices, it has become an increasing need to be able to update such models remotely. In industrial IoT context, this might be because the devices are cumbersome to get to, or due to the sheer amount which might be needed for an update, while the viability of FL significantly increases when not relying on a cable connection. OTA-TinyML by Sudharsan Et al.[28] addresses this issue, discussing the challenges of remote deployment

of models while introducing hardware friendly strategies for fetching and storing TinyML models from a cloud server in addition to supporting efficient loading and execution of said models. The TinyML models utilised are Tensorflow lite models which are stored as the serialization format FlatBuffers.

The challenges addressed by the paper are notable as they will serve as a bottleneck to overall commercial viability of TinyML as well as especially reduce the quality of FL due to the amount of communication required. They present the challenges at four levels. Firstly, the IoT hardware level deals with the heterogenous characteristics which need to be considered to ensure model compatibility and avoid bricking the device. Secondly, network and transport level challenges concern securing uninterrupted transmissions which requires the networks to scale and adapt to efficiently handle the traffic. Thirdly the cryptography level presents challenges due to the fact that optimized cryptographic libraries like TinyCrypt or Mbed TLS are required, so an appropriate needs to be selected, and they are less safe than the standard serverside cryptography strategies. Finally remote device management and operating system level is of concern as management systems need to be first and foremost secure, but also potentially compatible with shredded operating systems, for example Mbed OS or FreeRTOS. There are several suggested techniques for handling these issues. To solve hardware level issues, devices can be batched based on their hardware and software status, and can incrementally roll out updates. For ensuring secure channels for deploying OTA updates, channels like MQTT, which is thoroughly discussed later, can be used, in additon, data should be encrypted for protection, and physical security can be better ensured with tamper proof screws. Network congestion can be prevented by using a priority-based progressive deployment to ensure timely updates of model to handle the traffic. Authenticity can be ensured by having OEMs sign and hash firmware images for end-to-end integrity protection, devices can also verify models received by using public signing certificates and metadata. Finally, failure recovery should be possible by reverting to the previous model in case of a failed update, in addition, this should come in addition to enabling remote login and device log extraction.

In order to actually deploy the ML models, they concluded that the .h file containing the flatbuffer should be fetched from server using the http.get() method of the HTTOPClient object. Storage of the model takes place by reading the model in .bin format, thereafter allocating memory for it and copying the content byte-by-byte into SRAM during inference time. As such storing in binary format and securing SRAM storage is further proven to be the most optimal for microcontrollers.

3.2 Communication and aggregation for FL

A few works regarding general training strategies as well as federated learning strategies are addressed in this section.

3.2.1 General training strategies

In addition to the aforementioned advancements in on-device training, there have been advancements specifically in the area of the training both specifically in relation to, and independently from, resource-constrained devices which provide value to FL on IoT.

Challenges in federated learning, in general, include asynchronous participating devices and preventing malicious devices from participating, ”CoLearn”[8] by Feraudo, Angelo, et al. is a means to solve these issues in the context of IoT. It proposes the use of Manufacturer Usage Description (MUD) with IoT devices as well as a publish/subscribe system using a message querying telemetry transport (MQTT) broker. MUD aspect of the implementation assists in ensuring the safety of connected devices by having the MUD manager manage valid devices available for the centralized FL server, these devices are added to the whitelist through the devices themselves making a DHCP request to the MUD manager, authentication files will thereafter be stored in an external database. The architecture of CoLearn can be seen in Figure 4 specifying the communication between MUD implementation DHCP server, FL server, and devices. The issues concerning devices be-

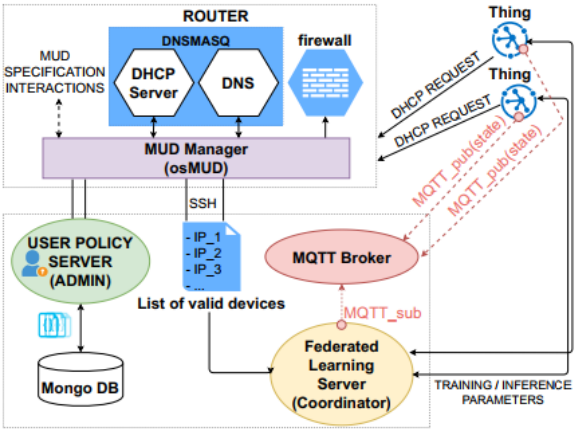


Figure 4: The architecture of CoLearn[10]. Note: ”Thing” refers to IoT device.

ing asynchronous are appeased using the aforementioned publish/subscribe pattern by dedicating the FL server to being coordinator (seen in figure 4) and having it subscribe to a specific topic for it to receive status updates from publishers, i.e. IoT devices, through an MQTT broker. Devices can have 3 states: Ready for training, inference on local data, or not ready. As such devices can publish when they are ready to participate in an FL round, allowing the FL server to use only available devices after a sufficient number are ready.

Though it provides a larger degree in the safety of operation the critical aspect of implementation viability is the additional communicative requirements, and the potential effect of only training on the devices declared to be

ready and not all. Aspects of implementation were tested using IoT BoTnet identification dataset and feed-forward neural network with 2 hidden layers of 50 and 40 neurons respectively. The loss decreases with the number of iterations but is mostly unaffected by the number of FL rounds which is attributed to the fact that the data used was independent and identically distributed on the devices. Another metric discussed by the authors of "CoLearn"[8] is temperature as it is desirable to stay under what might cause thermal throttling as well for saving electricity. In idle operation the device (Raspberry Pi 3 in this case), ran at 50°C, while 1000 and 3000 iterations increased the temperature by an average of 5.875°C and 6.75°C respectively, indicating temperature increases based on iterations illustrating another trade-off. The other area of significance is that it compared the aforementioned model training with a similar solution utilising Secure Multi-Party Computation (SMPC) privacy-protecting computational algorithm. The implementation using SMPC however proved to be inefficient as the training time increases far more significantly increasing batch sizes. The core value of CoLearn is however the demonstration of a more secure FL implementation that does not introduce an excessive burden on clients as it's designed with resource constraints in mind.

Designing with constraints of specific devices can be a burden due to the heterogeneity in available devices to use for a potential purpose while it's costly in terms of electricity and thereby CO2 emissions to train neural networks from scratch. A solution intended for this purpose is "Once-For-All"[3] (OFA) by Cai, Han, et al. which aims to train one network which is to allow for the extraction of sufficiently accurate sub-networks for the intended purpose and device from a larger OFA network without having to train the sub-networks, in turn accommodating for dynamic deployment environments. The implementation is a convolutional neural network trained on ImageNet, with a progressive shrinking (PS) algorithm assisting in maintaining accuracy. During training, it samples a few sub-networks with each update for training, but in order to reduce model size parameters are shared between the sub-networks resulting in only 7.7M, this means that interference between sub-networks is imminent during training. The PS algorithm is implemented to ensure that training is done to the largest network while later fine-tuning the smaller sub-networks as they are added to the sample, this minimises the interference of sub-network training on especially the larger ones, thereby improving accuracy significantly. The top1 ImageNet accuracy with 230M MACs using PS reached 76.0% while without PS and with 235M MACs reached 72.4%, and for comparison, a previous state-of-the-art hardware-aware neural architecture search (NAS) solution MobileNetV3-large had a top1 accuracy of 75.2% with 219M MACs. A core benefit of OFA is the constant GPU hour training cost independently of deployment scenarios (N), whereas MobileNetV3-large would require 180N GPU hours, training the OFA network takes 1200 GPU hours on 32 V100 GPUs. However for increased accuracy fine-tuning can be applied to specialised sub-networks in the OFA network which for 25 and 75 epochs increased the top1 accuracy to 76.4% and 76.9% respectively however with GPU hours now being 1200+25N and 1200+75N respectively. The details of OFA network performance in comparison to other state-of-the-art can be

Model	ImageNet Top1 (%)	MACs	Mobile latency	Search cost (GPU hours)	Training cost (GPU hours)	Total cost ($N = 40$)		
						GPU hours	CO_2e (lbs)	AWS cost
MobileNetV2 [31]	72.0	300M	66ms	0	$150N$	6k	1.7k	$18.4k
MobileNetV2 #1200	73.5	300M	66ms	0	$1200N$	48k	13.6k	$146.9k
NASNet-A [44]	74.0	564M	-	$48{,}000N$	-	1,920k	544.5k	$5875.2k
DARTS [25]	73.1	595M	-	$96N$	$250N$	14k	4.0k	$42.8k
MnasNet [33]	74.0	317M	70ms	$40{,}000N$	-	1,600k	453.8k	$4896.0k
FBNet-C [36]	74.9	375M	-	$216N$	$360N$	23k	6.5k	$70.4k
ProxylessNAS [4]	74.6	320M	71ms	$200N$	$300N$	20k	5.7k	$61.2k
SinglePathNAS [8]	74.7	328M	-	$288 + 24N$	$384N$	17k	4.8k	$52.0k
AutoSlim [38]	74.2	305M	63ms	180	$300N$	12k	3.4k	$36.7k
MobileNetV3-Large [15]	75.2	219M	58ms	-	$180N$	7.2k	1.8k	$22.2k
OFA w/o PS	72.4	235M	59ms	40	1200	1.2k	0.34k	$3.7k
OFA w/ PS	**76.0**	230M	58ms	40	1200	1.2k	0.34k	$3.7k
OFA w/ PS #25	**76.4**	230M	58ms	40	$1200 + 25N$	2.2k	0.62k	$6.7k
OFA w/ PS #75	**76.9**	230M	58ms	40	$1200 + 75N$	4.2k	1.2k	$13.0k
$\text{OFA}_{\text{Large}}$ w/ PS #75	**80.0**	595M	-	40	$1200 + 75N$	4.2k	1.2k	$13.0k

Figure 5: Various differences between variations of OFA and other previous hardware-aware methods on Pixel 11 phone[3]

seen in Figure 5. In addition, the CO2 emissions were lower than the previous state-of-the-art solutions where for example the OFA networks with no fine-tuning epochs produced 340lbs of CO2 compared to MobileNetV3-large producing 1800lbs when N = 40. Within the context of IoT suffering from potential resource constraints and heterogeneity in hardware, OFA can provide the benefit of efficiently producing multiple sets of suitable networks efficiently which can thereafter be part of their own continuous federated learning cycle with devices sharing sub-network. It can function independently in creating these models for immediate use in practical applications, or work as a starting point in for example the CoLearn[8] implementation to allow the independent devices to change their given network over time. These aspects are unexplored however as the focus is a better centralised learning strategy for different deployment scenarios, and the OFA solution might therefore suffer the privacy and practical drawbacks federated learning seeks to alleviate described in section **??**. It is still evidently valuable though, especially in its handling of heterogeneity, and the possibility of accurate small networks potentially able to fit on SRAM.

3.2.2 Federated Learning Optimizations

The process of ML model pruning is a mature means of compressing models by removing weight connections in neural networks, but not in the context of federated learning especially in conjunction with TinyML. Huang Et al. address this in their work on FedTiny[14], a pruning framework for pruning on the resource-constrained devices themselves as a means of fine-tuning to local data, while also attempting to alleviate biased pruning which might be caused by unseen heterogeneous data over devices by introducing batch normalization selection module.

The batch normalization selection takes place and updates the measurements for all candidate models before the evaluation so that a less biased initial coarse-pruned model can be selected before evaluation. The progressive pruning which takes place on the devices aim to further fine-tune the provided model by growing and pruning in accordance with average importance scores. The memory footprint and computational cost is thus reduced. Only top-K importance scores of the pruned parameters are maintained and the optimal structure is progressively approached. When deploying such a

pruning strategy not much accuracy was sacrificed on larger models, as it had only 1.5% lower accuracy than resnet18, however having a 1.5% better top-1 accuracy. It was however only tested on Raspberry Pi 3B making its potential in smaller microcontrollers less certain.

3.3 Compression for FL and IoT

Works concerning issues of deploying ML on tiny devices have been developed to solve both practical issues of enabling functionality, as well as optimisations of process improving all aspects of the existing implementation. As previously stated, power usage and memory usage go hand in hand in ensuring optimisation of both computational speed and viability of battery-powered remote devices.

In order to cope with the requirements of the model size described in section 2.4.1, so that model will fit on the device and preferably run on SRAM explained, Song Han, Huizi Mao, and William J. Dally proposed deep compression of neural networks (NN) using pruning, trained quantization, and Huffman coding, hereby referred to as simply deep compression[10]. Network pruning is the model compression technique where less significant weights/neurons are removed from the NN to reduce its size while maintaining the accuracy to a variable yet fair extent. Trained quantisation will further assist in compressing the NN by reducing the number of bits needed to represent the weights. Weight sharing is thereafter applied to each layer making all weights that fall into the same cluster share the same weights reducing the amount of trainable weights in the network. Finally, Huffman encoding is used to make common symbols represented with fewer bits with its lossless compression, this alone could reduce the network storage by 20%-30% performed on data set with previous compression methods applied. For further understanding of the individual effect, pruning on AlexNet resulted in $9X$ reduction in parameters, but can be more or less depending on the network, performing weight quantization thereafter was able to quantize to 8-bits, i.e. 256 shared weights, for all convolutional layers and 5-bits for fully connected layers with no accuracy loss. When using the described deep compression on AlexNet, the network size was reduced from 240MB to 6.9MB, i.e. $35X$ compression rate to 2.87% of original accuracy. The goal of this research was to improve the capabilities of networks running on mobile applications, meaning that after compression the AlexNet would fit on the SRAM on relevant devices. However, as SRAM on microcontrollers is mostly 512KB or less[29], the need for effective compression and therefore the value of deep compression becomes more evident. The necessity is visible in the aforementioned battery constraints of microcontrollers, as for example under 45nm CMOS technology 32bit SRAM memory access takes 5pJ while DRAM requires 640pJ, meaning that running a 1 billion connection neural network at 20 frames per second (Hz) would require 12.8 Watts using DRAM while SRAM would require 0.1 watts[10].

The accuracy of the networks after different compression strategies were

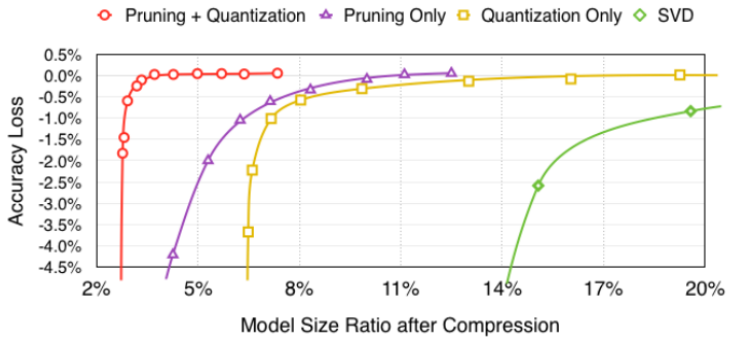


Figure 6: Deep compression: Accuracy loss based on model size ratio and compression methods performed on AlexNet[10]. Also showing SVD, another independent method of compression.

shown to generally decrease after the 8% mark as shown in figure 6 which was performed on AlexNet. However, it becomes apparent that pruning and weight quantization together have less of an effect on accuracy when combined than when used individually in the context of this experiment, while simultaneously decreasing network size. The justification for weight quantization's positive effect on pruned networks is because given the same amount of centroids the quantization causes less loss on networks with fewer weights, visible in the aforementioned figure as AlexNet goes from 60 million weights to 6.7 million weights post pruning. Deep compression can therefore be viewed as a particularly effective network compression strategy while it is evident the accuracy cost is unlikely to inhibit the quality of service. As such, even if it's intended for mobile applications it is applicable for the networks meant for more constrained devices as well, however, the degree to which it is able to compress networks is not too consistent, as it saw a range of $35X$ to $49X$ compression rate independently of initial size in MB. It is therefore still relevant to train the initial network with its size in mind to eventually reduce it to less than 512kb to fit on SRAM.

3.4 The situation as a whole

Despite the potential benefits of on-device training for microcontrollers, the adoption and study of this approach have been limited. As highlighted in previous sections, the primary reasons is the resource constraints of microcontrollers, which typically possess limited computational power, memory, and energy reserves. These constraints pose significant challenges in implementing complex machine learning algorithms, particularly deep learning models that require substantial computational resources. Additionally, the heterogeneity of microcontroller hardware and the absence of standardized platforms make it difficult to develop generalizable solutions that can be widely adopted across different devices.

Moreover, the existing research on machine learning for microcontrollers has primarily focused on efficient inference rather than training. This is be-

cause inference is more frequently employed in real-world applications, so optimizing for it yields more practical benefits. Furthermore, the current research on on-device training has predominantly centered around more powerful devices, such as smartphones and edge devices, as they offer a better balance between resource availability and mobility. Consequently, the limited research focus on microcontroller-based on-device training has led to a dearth of mature tools, libraries, and frameworks that can facilitate its adoption.

As for federated learning in the context of on-device training, the scarcity of studies can be attributed to several factors. First, federated learning was originally proposed as a means to address privacy concerns in the context of centralized training, where the primary objective is to aggregate and learn from decentralized data without exposing individual data points. This motivation is less relevant in the context of on-device training, where the learning process occurs locally, inherently preserving data privacy.

Second, the communication overhead associated with federated learning can be substantial, particularly when dealing with microcontrollers. These devices have limited bandwidth and energy resources, which could be quickly depleted by the frequent exchange of model updates necessary for federated learning. This can be especially problematic in settings with intermittent or unreliable network connectivity, further hindering the adoption of federated learning in on-device training contexts. Lastly, the lack of a well-established ecosystem for on-device training in microcontrollers also extends to federated learning. Without the necessary tools, libraries, and frameworks, researchers and practitioners are less likely to explore federated learning for on-device training on microcontrollers. This, in turn, results in a limited understanding of its potential benefits and challenges, impeding its widespread adoption and further study. In the discussed related works, it is evident that individual aspects regarding on device training, efficient training and communication, and means to reduce memory and computational cost of ML have been understood, but experiments combining all aspects sufficiently is especially lacking. As such the proposal of this thesis is to provide guidelines on best practices on means to leverage on-device training optimization in conjunction with FL and compression strategies

4 Proposed Framework

As the benefits of TinyML have been made evident, and the advances in the fields of on-device training, federated learning, and model compression have taken place it is clear that real-world applicability is imminent. These technologies enable low-power, resource-constrained devices to perform complex tasks while preserving user privacy and reducing data transfer overhead. In this proposal, we present an approach for deploying and training fully trainable models for on-device training on microcontrollers, and how they can best benefit from the strengths of Federated learning to mitigate issues regarding their resource constraints, talked about in section 2.4.1. The framework deals with four key aspects of model deployment and training to be able to develop each core functionality with regard to each other. Firstly, compression methods for a centralized model in order to better fit on microcontrollers are explored, secondly, the on-device training is developed to function independently of FL while being compatible with the compressed model and deploying necessary techniques for improving the general training, thirdly, federated learning is added to the context to further understand the potentials in model improvement based on distributions of device and data and how weight averaging can be handled for best result, and finally the strategies for enabling more efficient communication for federated learning is addressed to further improve the context as a whole. Our solution aims to empower edge devices with the ability to adapt and learn in real time, thereby enhancing their performance and robustness while maintaining resource efficiency. This section provides an overview of the proposed implementation, justifications for decisions in the development process, its significance, and its potential impact on the broader machine-learning landscape.

4.1 Proposal

The proposal of this thesis is a systematic approach to principles in approaching implementing Tiny and Federated machine learning and thus serves as a framework for the readers to best develop the cooperating nodes, especially with a particular focus on RNNs using LSTM. LSTMs are the primary concern as the implementation of on-device training of such layers or even RNN architecture is severely underrepresented in the field, which is reflected by the lack of works related to it in the related works section, as such, investigating it specifically becomes a core contribution. This proposal includes the suggested practice of handling model compression of a pre-trained model in preparation for on-device deployment and FL, but also general compression of both models, communication between device and server, as well as means to reduce unnecessary weight averaging on the server. The proposed fully trainable models on microcontrollers are designed to have standard architecture and layers commonly found in larger models for FL compatibility with Tensorflow models, while exploring the potential of FL, as well as compression-aware techniques incorporated and adjustable by the developer with the purpose of assisting in lowering communication cost, speeding up computation, and improve convergence

by only transferring that of significance, and end up with a best possible final model for inference. In regards to the compression techniques, in particular, practical implementation and testing of quantization-aware training capability on microcontrollers were conducted to assess its viability in smaller models and specifically on LSTMs. This is done as quantization is the eventual final step taken before the deployment meaning that training in preparation for it should be done to decrease the accuracy loss. In addition, the communication cost decrease of routinely using quantized 8-bit integers instead of 32-bit floats is desirable. The aspects of value the proposal can be summarised as follows

Serverside PRE-Fl compression As inference tends to be the standard goal of the use of ML with IoT devices, several compression techniques which can be done on servers are tested with the purpose of assessing their potential effect when the goal is deployment on microcontrollers. In addition, knowledge distillation is presented as an especially significant compression strategy in this context, as it shows the most potential particularly because of its potential in use with the conventional strategies for tinyML model compression.

FL communication cost reduction The effects of various means of attempting to reduce the communication cost with the server are addressed in relation to figures generated in other test cases. These include technical aspects of on-device training as well as systematic decisions in the learning process and the consequences.

On device training Aspects of the training process can be adjusted and considered when developing on-device training individually, as well as what is good practice when it is in a federated learning context. The parameters which degrade performance are discussed to assess potential means of mitigating the loss of precision, while the

4.2 Library and system architecture

There are two core parts that the system is divided into, namely end devices and edge servers. The end devices consist of several Arduino nano 33 BLE Sense, and Sparkfun Edge Apollo 3, both with 1MB flash storage and 256KB and 384KB of SRAM respectively. The second layer consists of the edge server which is connected to the end devices through a serial port. As end devices train on their available dataset, their local weights are routinely sent to the edge server, the edge server then runs federated averaging to create new weights and redistributes the new and updated weights. Finally, the edge servers are connected to can be connected to a cloud server which functions as primarily a means to gather and process data but also could have the potential to run FL on multiple edge nodes. In a practical test bed, this was done using my computer as a sort of cloud, with a raspberry pi 3 node serving as the edge node connected to edge devices through usb. The implementation was however impractical as I only had one Raspberry pi 3 with limited USB ports, so when testing on the microcontrollers

my computer served the purpose of both cloud and server. When the system is first initialized, the cloud server could be responsible for running various compression algorithms including quantization and teacher-student compression, to downsize the larger pre-trained model to a predetermined size pre-configured on the edge servers and end devices. However, as the main goal is testing on-device training specifically, the layers above, in particular edge, is focused primarily on the federated averaging of newly initialized models on the device. This will be done using CNNs and especially LSTMs to cover a wider set of use cases for a more generalized understanding, while DNNs are also addressed briefly as it is a simple implementation next to the two other architectures.

4.3 On-Device implementation

Using a simple self-made c++ library developed for on-device learning on Arduino and Sparkfun, models can be implemented easily by stacking the necessary layers and connecting the relevant input and output. An example of this can be seen in the code listing 1. To ensure full control of memory expenditure, variables and arrays that need dynamic allocation are pointer arrays and the necessary computation on such vectors are handled by extensive use of loops. This is relatively slow compared to the backend of typical libraries which handles these operations but gives full control over memory usage without overhead.

An example of such a matrix calculation using said pointer arrays can be seen in the loop calculating the gates of the a cell in the model in listing 2. While likely an insignificant factor to the overall performance of the model, it also shows loop unrolling, as weights are indexed using an enum instead of the additional loop. When talking about LSTM specifically the terms ”unit” and ”hidden size” are used interchangeably, but in the context of this task it means both the number of hidden weights, as well as the number of cells in the layer.

4.4 Data used

The data used in this project is the CIFAR10 image dataset for CNN and DNN training, and air quality data from Norsk Institutt for luftforskning (NILU) for the LSTM training. The CIFAR10 dataset is widely used in computer vision ML model benchmarking and consists of 60000 32x32-pixel RGB images uniformly distributed across ten classes: Airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Due to its small size, diversity of classes, and equal number of samples per class, it ensures balanced distribution for accurate evaluation and thus serves as a solid dataset for comparing various types of models from traditional approaches like Support Vector Machine to advanced ones like CNN.
The air quality data for testing LSTM was generated by (NILU) and contains the measurement of PM10.0 data from one of 15 locations in Oslo with timestamp intervals of every 15 seconds between the date and time 2022-05-16 13:09:15 and 2022-05-27 11:01:45 in the form of a CSV[20]. The content of the CSV file consists only of the time stamp and PM10.0 measurement at said time, when exporting data from the websites, quality as-

Listing 1 Implementation of a simple CNN forward and back propagation sequence

```
//Forward
convl1->forward(conv_input, conv_output);                                        // Conv2d layer
maxpool1->forward(convl1->filters, conv_output, pool_output);                    // Maxpool2d layer
flatten(convl1->filters, pool_output, mpoutsize, 1, model->batch, f_output); // Flatten
model->forward(f_output, output);                                                // Dense layer

l2_error=lambda*L2(model->weights, model->in_sz, model->out_sz, model->batch)/2;
error=cross_entropy_loss(output, ground_truth, model->batch, model->out_sz)+l2_error;
acc=accuracy(output, ground_truth, model->batch, model->out_sz);


if(verbose == true){
    if(local == true){
        cout << "Softmax ";
        for(int y=0; y < model->out_sz; y++){
            cout << output[0][y] << " ";
        }
        cout << endl;
        cout << "\tLoss " << error << "\n";
        loss_list[epi] = error;
    }else{
        Serial.print("\tsoftmax");
        Serial.println(output[0][0]);
        Serial.println(output[0][1]);
        for(int y=0; y < model->out_sz; y++){
            Serial.println(output[0][y]);
        }
        Serial.println(error);
    }
}

if(epi == local_episodes-1){
    cout << "Train Accuracy: " << acc << std::endl;
}

//Backward
model->backward(output, ground_truth, loss, f_output, LR, lambda);
inverse_flatten(convl1->filters, loss, mpoutsize, 1, model->batch, gradients);
maxpool1->backward(gradients, inverse_maxpool);
convl1->backward(conv_output, conv_input, inverse_maxpool, LR);
```

Listing 2 The gate calculations within the LSTM cell

```
for(int i = 0; i < hidden; i++){
    sum_i = 0;
    sum_f = 0;
    sum_o = 0;
    sum_c_ = 0;

    for(int j = 0; j < input_size; j++){
        sum_i += weights[IFCO::I][j][i] * x_t;
        sum_f += weights[IFCO::F][j][i] * x_t;
        sum_o += weights[IFCO::O][j][i] * x_t;
        sum_c_ += weights[IFCO::C][j][i] * x_t;
    }
    for(int j = 0; j < hidden; j++){
        sum_i += hidden_weights[IFCO::I][j][i] * prev_h[0][i];
        sum_f += hidden_weights[IFCO::F][j][i] * prev_h[0][i];
        sum_o += hidden_weights[IFCO::O][j][i] * prev_h[0][i];
        sum_c_ += hidden_weights[IFCO::C][j][i] * prev_h[0][i];
    }
    sum_i += biases[IFCO::I][i];
    sum_f += biases[IFCO::F][i];
    sum_o += biases[IFCO::O][i];
    sum_c_ += biases[IFCO::C][i];


    i_t[0][i] = sigmoid(sum_i);
    f_t[0][i] = sigmoid(sum_f);
    o_t[0][i] = sigmoid(sum_o);
    _c[0][i] = tanh_func(sum_c_);

}

```

surance, and quality control values follow to indicate the overall value of the data, but these have been filtered out of the file.
As the end devices used in the project do not possess the capability of gathering such data themselves they need to be loaded onto the device before starting the training process. As such the data has to be formatted to respect the resource constraints present. This is done by formatting the data into arrays of the desired input size, which in this case is (12, 1), and scaling the numbers on a range from -1 to 1. This scaling is done as ensuring that all input features are on a similar scale should help the model converge faster, in addition, the tanh and sigmoid activation functions used in the LSTM cell gates also have ranges of -1 to 1 and 0 to 1, respectively, as such if the input data is much larger or smaller than the range of these activations, the consequence might be vanishing or exploding gradients. The data is thereafter loaded onto the device to be used in the training phase, by default each device gets 500 datasamples each for their dataset unless stated otherwise in the evaluation section.

4.5 Models Used

Due to the goal of testing a greater variety of tinyML with FL use-cases, compatibility with three different NNs was implemented in favor of focusing on one alone. DNN was chosen due to the fact that it is a component of other neural networks and can function as the sole trained layer in a greater network, but also because of its general use in multiple problems and low complexity. CNN is the standard for image classification and with the need for surveillance over people and industrial machinery, having access to training in computer vision on resource-constrained microcontrollers could be beneficial to ensure safety in dangerous work areas. Finally, LSTM is used for time series prediction in cases where predicting the future based on previous events in their specific time sequence is necessary. In an industrial context, LSTM could be beneficial for detecting developing anomalies in machinery based on several data, for example, familiar pressure and temperature changes over a set timeframe might indicate imminent failure.

4.5.1 Deep Neural Network

The implementation of DNN capability is a natural addition as the core component, the FC layer, tends to be the final layer of several other NN architectures, this was demonstrated in the Related Works section by both TinyfedTL[15], and TinyOL[24] where they deploy a trainable FC layer at the end of a static Tensorflow Lite model, while also showing promise independently like in the work by Grau et al.[17] where a full DNN was deployable. Whereas TinyFedTL and TinyOL were intended for image recognition, Grau et al used their DNN for voice recognition, in addition, it can be viable in text classification and tabular data analysis. As it works as a jack of all trades, but master of none, it is clear that when exploring the potential of TinyML, other architectures more suitable for specific tasks are also necessary which is why this work focuses primarily on the other two NNs.

As described in the background section, DNNs are fully connected, meaning that the weights and biases per layer would equate to input_size_*output_size+output_size. This means that the feature extraction is on the context as a whole and that consequently the weights needed to achieve similar accuracy as more advanced architectures come at the cost of scaling up layer sizes or alternatively increasing the number of layers. For example, for image recognition, DNNs lack the ability to handle spatial information or local patterns whereas CNN can utilize filters for capturing these local spatial patterns and take advantage of the hierarchical nature of images, while also using the filters for weight-sharing. As such DNN becomes competitive by increasing complexity against an architecture with inherently lower demands. However, based on the complexity of the task, DNNs might still be sufficient in instances with non-spatial and non-temporal data, or in instances where feature extraction has taken place prior to the DNN, and can achieve comparable results to other models if input data is less complex.

As a result, the implementation of DNNs is natural for testing a common use case for ML in general, while the simple architecture means there is little room for architectural change in the pursuit of a faster or smaller model. This makes DNNs good for benchmarking and gaining insight into the performance and optimization of other architectures.

4.5.2 Convolutional Neural Network

Convolutional Neural Networks are necessary for testing the limits of microcontroller on-device training for computer vision as it is the convolutional layer that enables the capturing of local spatial information for feature extraction. Based on CNNs better architectures have emerged like ResNets[11], however, these require an increase in parameters and memory consumption in both training and inference. As such, standard CNNs are sufficient in testing image recognition as the primary goal is to achieve fast viable models with low memory usage and efficient communication with the server, making the more advanced alternatives that are known to require more parameters less viable. ResNets' higher parameter requirements stem from their skip connections and additionally required layers, which decrease their scalability and make them less attractive for microcontrollers. Nevertheless, with convolutional layers viable, extending the class to allow for residual connections can be done if desired, however, conforming to this architecture might negatively impact the customizability of the model architecture as the required additions remove from other alternative layers which might be more suitable when designing NN model for specific use-cases.

4.5.3 RNN: Long Short-Term Memory

For the purpose of handling sequential data, like time-series data or natural language processing tasks LSTM is a suitable candidate architecture due to several reasons. Previous RNN structures were insufficient at handling long-range dependencies in sequential data due to them suffering from vanishing and exploding gradients, however, LSTMs mitigate the issue through a gating mechanism. This mechanism enables LSTMs to main-

tain information across longer sequences while selectively remembering and forgetting, which allows them to capture long-range dependencies in addition to being more robust to noise in the input data. The versatility due to improved handling of data is therefore especially valued since when intended to deploy on microcontrollers, the variability in real-world data can impact the performance significantly as the training set might not be entirely representative as shown in Haoyu et al.'s TinyOL[24]. Microcontrollers will possess less data-processing due to resource constraints, inherently making it subject to more noise in the data-set, while in addition, the intention of making use of the models in an industrial IoT FL scenario makes the data more vulnerable to a larger number of variations in environment, with adjacent machinery potentially impacting the sensors differently. Thus, better long-term feature extraction is necessary for gathering the desired features efficiently, while forgetting what features learned which are deemed unnecessary. Therefore, the federated learning process can also become like a federated forgetting process for what they collectively want to disregard when training.

While LSTMs are still a popular choice in general when dealing with time-series data, transformers become the new state-of-the-art in handling sequence-based tasks, however, there are core differences that make LSTMs more desirable for microcontrollers and FL specifically. Firstly, the reduced complexity increases parameter and energy efficiency as fewer weights are needed to be kept which transformers need for the attention mechanism and additionally required layers. In a scenario where the LSTM input-to-hidden, hidden-to-hidden and cell-to-hidden weights are 512, the LSTM would contain 784432 weights whereas a single layer transformer of similar size would require 33% more for the self-attention layer alone, while input-to-hidden and hidden-to-output weights are also required, which can increase the requirement significantly based on the hidden size and neurons for FC layer. In addition to the size issue of transformers, however, LSTMs are better for incremental training whether online or normal incremental as they adapt to new data in a sequence. Transformers require entire sequence to be available at once for training, making it less suitable for scenarios where data is streamed or where it is undesirable to store long data sequences.

As a result, LSTM is favored as a good architecture for microcontrollers due to their size, inherent processing of data, and increased compatibility with incremental and therefore online and federated learning.

4.6 Training Processes: Federated and Central on-device

In order to compare the results from the different models present in evaluation, models were tested by running both federated learning and training alone on their own dataset.

4.7 Server-Side weight averaging

The server-side weight averaging during federated learning was done using a standard federated averaging algorithm which adds all the weights from each device and averages it. The implementation of the averaging waits

for all devices to finish their local episodes before performing the averaging, as it gives a more balanced overview of performance and devices an equal impact on the weights. The default training parameters for federated learning in this implementation it that there are 24 federated rounds, with each device having 5 local episodes, however, alternatives are explored.

4.8 Measurement of Metrics

Each separate device calculates its loss based on its training data during its training process, in the case of LSTM, the MSE loss is utilized. After each device has finished training, each device is tested through the validation dataset before the federated averaging happens to obtain the MAPE and MAE scores

4.9 Model Compression before and during FL

In the context of federated learning, server-side compression plays a critical role in addressing communication overhead and bandwidth limitations, which are particularly crucial when dealing with resource-constrained devices. By applying compression techniques before or during federated learning, it is possible to reduce the amount of data exchanged between the server and the participating devices, thereby minimizing communication latency and preserving the energy resources of the devices. As such, testing the overall viability of compression whether with the purpose of creating a good pre-trained model for fine-tuning using on-device training and FL or just in order to deploy a finalized model for on-device inference alone is necessary in order to gauge the value of the on-device training. As some degree of compression is inevitable when the model is done training centralized, or federated, some understanding of how this process

Server-side compression can be employed before federated learning to reduce the size of the initial model that is transmitted to the participating devices. By compressing the model, the server can ensure more efficient distribution of the model to the devices, allowing them to begin the local training process sooner. Several model compression techniques, such as weight pruning, quantization, and knowledge distillation, can be applied to achieve this goal. These techniques focus on reducing the redundancy in model parameters, lowering the precision of the weights, or transferring knowledge from a larger model to a smaller one, respectively, while maintaining the model's performance.

Updates can be done during the federated learning process which acts as a compression of the FL process itself by compressing the model as usual but also optimizing the algorithm to better handle the weights transmitted from the devices, improving the weight averaging process speed and convergence rate. This usually consists of gradients or weight deltas and can be compressed using techniques like lossy compression, sparsification, or sketching algorithms. Lossy compression reduces the precision of the model updates but maintains their overall structure, whereas sparsification involves retaining only a subset of the most significant updates. Sketching algorithms, on the other hand, provide compact representations of the updates, allowing the server to reconstruct an approximation of the original

data. By compressing the updates, the server can reduce communication overhead while aggregating the updates from multiple devices.

As such, server-side compression plays a pivotal role in optimizing the federated learning process, particularly when working with resource-constrained devices. By compressing the model and the updates before and during federated learning, it is possible to mitigate communication overhead, reduce latency, and conserve energy resources, ultimately leading to a more efficient and scalable federated learning system.

4.9.1 Before On-Device Deployment

Various techniques have been employed to achieve the best possible tinyML model before FL training, including weight quantization, pruning, and weight clustering with Huffman encoding. As discussed in section 2.5, Weight quantization involves reducing the number of bits used to represent the weights in the model, thus decreasing the overall memory requirements. Pruning, on the other hand, eliminates redundant or less significant connections within the network, leading to a sparser model with fewer parameters. Huffman encoding further compresses the pruned and quantized model by employing a variable-length coding scheme based on the frequency of weight values, enabling more efficient storage.

One of the particular, most promising, approach for model compression is the teacher-student paradigm, where a smaller student model is trained to mimic the behavior of a larger, more accurate teacher model. The student model learns from the teacher's outputs or intermediate representations rather than directly from the ground truth labels, effectively transferring the knowledge from the teacher model to the student model. This process allows the student model to achieve a high level of accuracy while being more compact and computationally efficient, making it well-suited for deployment on resource-constrained microcontrollers. By combining the teacher-student paradigm with other compression techniques, such as weight quantization, pruning, and Huffman encoding, it is possible to create highly efficient, low-memory footprint models that can effectively operate on microcontrollers while maintaining satisfactory performance.

4.9.2 During Federating Learning

The deployment of machine learning models on microcontrollers demands the use of compression methods not only after the training process but also during training, to ensure the resulting models are both efficient and accurate. Quantization-aware training (QAT) is one such technique that incorporates quantization into the training phase itself, allowing the model to learn to adapt to the reduced numerical precision of weights and activations. By simulating the effects of quantization during training, QAT ensures that the model's performance remains robust even after the weights are quantized for deployment on microcontrollers.

4.10 On-device optimizations

Optimization of the on-device training process includes a few key parts which are explored in this thesis. The

Compact architectures Different model sizes were tested to assess how the number of weights might affect the different performance metrics of the model as a whole. The standard LSTM model has 16 units, which is significant for both how many weights the layer has, as well as how many cells the layer possesses for the data to pass through. In addition, a smaller model of 8 units was tested for comparison and a larger model of 64 units. The units become significant for the overall speed of the models as the increases in it will rapidly require more storage for weights which means more values for matrix multiplication, while also adding another entire computation step for each unit through the cell.

Quantization-aware training Quantization-aware training is briefly explored to address the concerns of performance loss whether for whether when the model is finally quantized for inference, or if it would be desired to quantize the model for weight transfers.

4.11 Unaddressed Work

This section is for planned work which I wanted to complete but could not due to low gain, and the amount of work necessary to implement other factors.

4.11.1 Power consumption measurements

I attempted to benchmark the power consumption of various deployments of models on an Arduino and attempted to use two separate USB multimeters of different brands to read the current, but it was unable to read the low current. The benefit of having this metric is to further see the effect of different storage of different variables on the device, i.e. the effects of locating them on SRAM vs flash memory. As it could not be measured, the implementations were made entirely with the goal of fitting on SRAM with enough complexity to be able to provide value.

4.11.2 Model or data on Flash memory vs SRAM

As having especially the NN model on SRAM is significant for computational speed on microcontrollers[10], benchmarking the consequences of utilising the flash memory just for the sake of benchmarking is unnecessary for the task. This is especially relevant as strategies for using the flash memory are primarily through constants, meaning that even in a real-world scenario with continuous data sampling it would not be very viable for storing ever-changing datasets either, while in addition, very little SRAM should be needed for a sufficient batch variable to store the streamed data temporarily before deleting it. The concern of flash memory is more

relevant when concerned with inference as the weights in that case are static and have the potential to be placed in flash.

5 Evaluation

The evaluation section will cover experiments on different system variables to find ones that provide sufficient performance to the models being trained. It will address experiments on compression for inference, the behavior of on-device training based on the various parameters, and the effects of training strategies commonly employed server-side instead used in on-device to investigate the potentials. While DNNs and CNNs are briefly addressed, the focus is on RNNs using LSTMs. All the graphs present are created using python with Matplotlib for the plotting.

5.1 Set-up

To run the tests used to generate the data in the evaluation, both simulations and a real-world test bed were used. The simulations were scripted in C++ to be entirely compatible with the code which was deployed on the device and serves the purpose of better exploring the effects of various parameters which the test bed cant cover, e.g. having more devices available. The test bed consists of a maximum of 3 Arduino Nano 33 BLE Sense devices, and 2 Sparkfun Edge Apollo 3 devices, connected to a computer that handles the federated learning process using a Python script, through a USB cable for serial port communication. The computer that is used to record the performance has an ASUS GeForce RTX 2080S GPU with 6 GB memory, an Intel Core i9-10900 CPU at 2.80GHz, 32 GB RAM, and running Windows 11.

For reproducibility to allow for more extensive testing on LSTM scenarios, the models are initialized with same weights using a stored set of Xavier/Glorot initialized weights as they work well with networks with sigmoid activation[9].

5.2 Pre-FL Model Compression

To see how good a couple of relevant, well-performing, models could become after applying compression, some evaluations of compression strategies that would take place before distributing the model for FL or Inference is needed.

5.2.1 CIFAR10 Image Classification

Methods for compression before the federated learning process takes place to discover what server-side compression can be applied to the pre-trained model to garner the best result in the least accuracy loss compared to model size reduction. In particular, the knowledge distillation technique is core before the FL rounds take place as it is a very effective way of creating a much smaller student model with similar accuracy as the larger teacher model[4]. As we assume that the model architecture and weights will be identical on all devices participating in the federated learning and therefore that the dimensions of the weights will remain the same, knowledge distillation becomes the natural first step to preparing the model for devices

if a pre-trained model is desired for fine-tuning or inference. Alternatively, a small model with a similar size to the student model can be initialized with random weights and trained on the same data used by the teacher-student strategy, however, this tends to yield lower performance[12]. As such when deciding on a model to use for Tiny and Federated ML, firstly the teacher model is trained, thereafter a smaller model of similar architecture is made, though not necessarily completely similar, and knowledge distillation can take place on said model. Finally, other compression algorithms which can be done both before FL starts and during the process can be applied, i.e. pruning, quantization, and weight clustering, which were addressed in deep-compression[10]. To illustrate the effect of each, figure 1 shows their use alone and in various combinations on an alexnet trained on CIFAR10, where each of the students' layers has 25% of the units the teacher has. This demonstrates how effective the combination of all four can be, as an accuracy loss of 1.2% is the only price to pay for a 99.3% model size reduction.

Model	**Size**	**Accuracy**
gzipped Keras model	77329	0.698
Pruned-TFlite	2827639	0.708
Quantized-TFlite	232874	0.699
Distilled-TFlite	2827639	0.708
Pruned, Quantized-TFlite	10761290	0.710
Distilled, Pruned, Quantized-TFlite	10763536	0.710

Table 1: Result from various compression strategies and their combination on an AlexNet. Each layer in the student has 25% of the units of the teacher's layers.

Based on the figure, it is clear that knowledge distillation, particularly on, and especially in conjunction with others, is sufficient for creating significantly accurate models. When attempting to train the distilled, pruned, quantized model from scratch, reaching a similar point with freshly initialized weights proved to be unlikely as a model with lower weights will struggle to gain the generalizations a larger model would.

5.2.2 PM10.0 Time-series classification

Secondly, in order to see the potential of smaller and, compressed LSTM models on time-series data, a roughly 49,985-weight 2-LSTM layer model was made and trained, then distilled into a model with 329 weights. The resulting compared prediction is visible in figure 7 where the original LSTM had MAPE of 60.18% while the distilled model had a MAPE range of roughly 60%+/-2%, with the case in the image figure slightly smaller MAPE at 58.69%. Such a MAPE is generally not considered, too good but as has been mentioned and will be seen in later figures, it likely happens due to the predictions being on a range from 0.0 to 1.0, in which MAPE might struggle. However, the general trend of the true data is followed by both models, indicating that the model is at least viable to some degree

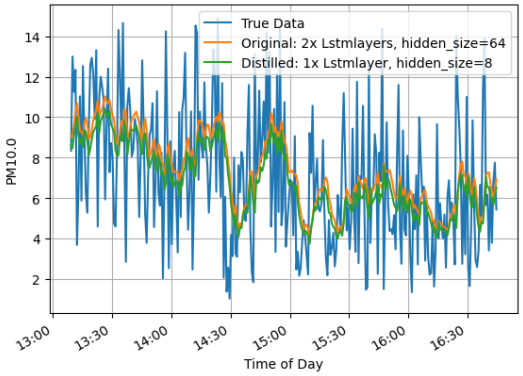


Figure 7: Original two-layer 64-unit RNN with LSTM vs Distilled, compared to the true data

5.3 Evaluation of Neural Networks on Microcontrollers

The general viability of the models for on-device training is discussed with examples demonstrated for on for the implications of on-device training. As both CNNs and DNNs are relatively mature architectures in this context, the general comparison and contrast here and in later sections will be focused primarily on the LSTM model as it is the area with the least FL and tinyML implementations, as I have not any relevant articles for it, contrary to the two others. DNNs and CNNs are however briefly assessed.

5.3.1 Deep Neural Network

Standard deep neural networks generally require more parameters for less performance than what a CNN usually does, so on its own it performed sub-par as it would have to eat up more of the space which is needed for data especially in image classification. No result of significance was yielded alone which the two other models outperform it in the scenarios with the dataset used. While not very viable for on-device in its entirety, naturally both other models use the core component of a dense layer at the end to process the output, whether classification or regression, so it serves that purpose well as known from standard ML practice.

5.3.2 Convolutional Neural Network

The model briefly testes here was a CNN consisting of a convolutional layer with 16 filters and 3x3x3 kernel size for RBG channels, thereafter a max-pooling with 2x2 kernel, flattening and a dense layer. The model size is therefore small enough to easily fit on data, which is necessary for to fulfill the immense memory requirement that the image dataset will place on the model. As seen in figure 8 the loss curve is smooth as it is converging neatly, and the accuracy signifies it training, albeit not very well. There is plenty of room for improvement on accuracy, but the converging reaching its minima indicates that a greater dataset is needed. The data aforementioned data requirement however poses the greatest resistance to this as an image of

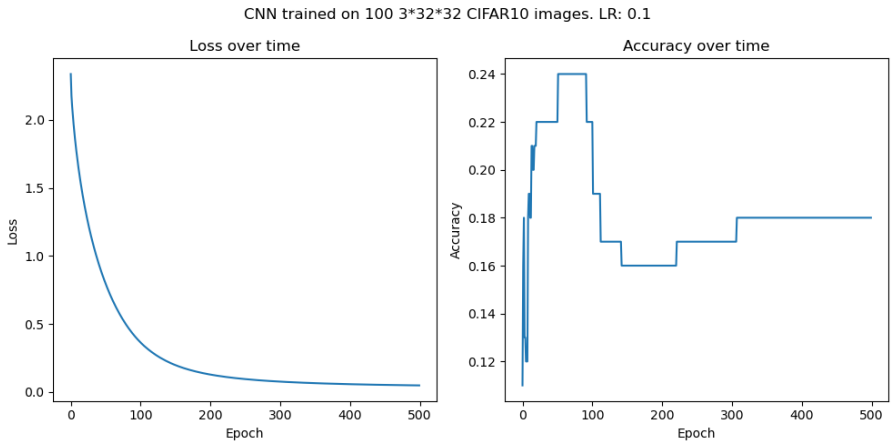
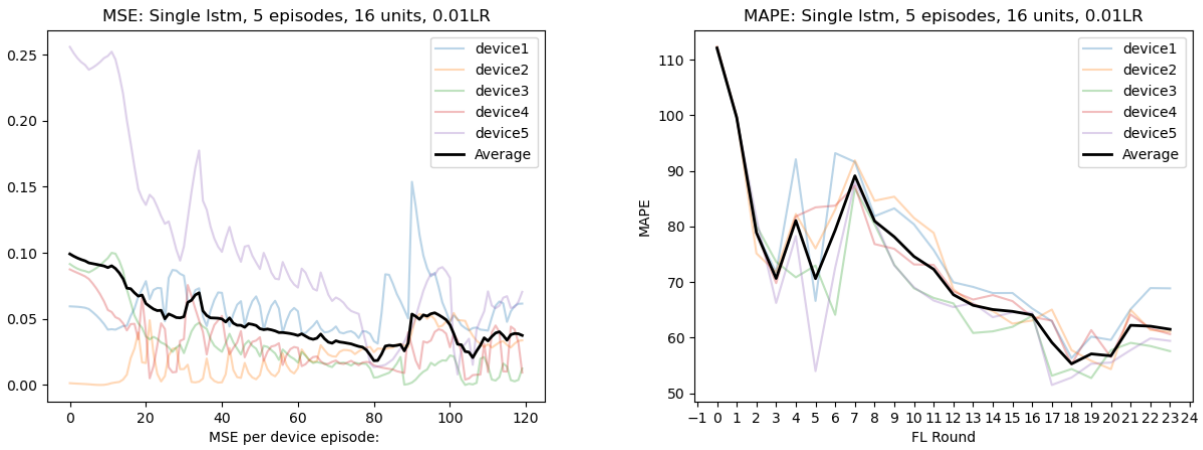


Figure 8: Loss and accuracy of a CNN trained on a single device

size 32x32x3 than 13.5KB, meaning that when relying on storage of data samples instead of streaming or self-sampling the data will always be minimal and which can make FL training less fruitful as the devices doesn't have access to learn enough common generalizations, this is discussed further in section 5.5.3. And it can be assumed that generally, the model will face similar problems as the big LSTM model in figure 14, whose devices

5.3.3 Long Short-Term Memory

Lstm was implemented firstly with the intention of only having one layer which could be sufficiently good, which is followed by the dense layer. This is because the significant amount of output generated by the LSTM with return sequences set to true, as well as the second layer having a need for the storage of its own intermediate variables would place a significantly greater burden on the device in terms of computation time and storage.



(a) The MSE

(b) The MAPE

Figure 9: 64 Units LSTM training and validation results of 5 devices trained with FL. Each has 100 data samples to train with.

Figure 9 illustrates that a relatively low amount of weights on a single layer can demonstrate a relatively well-improving performance on time series prediction when considering just the MSE loss metric, and how federated learning is viable for this. The model has a 16-unit layer taking 12 inputs, and a dense layer with a single output, this equates to 1152 weights, which when using 32-bit floats means it requires 4.56KB, with an additional 0.7KB per unit for backpropagation intermediate variables, which is a total of 16.06KB, 6.2% of the SRAM limit posed by an Arduino. It is noteworthy that after every FL weight averaging it increases the loss as generalizations inappropriate for independent devices are gained, however, the overall performance improves consistently. This is particularly noticeable with ”device1” whose loss progression indicates that the federated averaging of the weights is generally to the detriment of its individual model as it jumps significantly. A notable observation is how the increases in MSE whether at the start of each FL round or in the middle of FL rounds are not reflected in the MAPE score. The cause for this deviation of metrics which are both supposed to represent the quality progression during the training process is likely the fact that they emphasize different aspects of the error. As stated before, while MSE is sensitive to large errors, i.e. through having few but large outliers, the MAPE deals with average values, which are naturally less affected by outliers. As such The MAPE is a better generalized indicator of how the progression of how the training is proceeding. Therefore, based on figure 9, it can be assumed that the spikes presented in the MSE by weight averaging are not detrimental to the average performance. The best average MAPE presented indicates a $\tilde{5}6\%$ deviation from the desired output on average, which signifies the accuracy of the model being very low, however, some of this could be attributed to the prediction range being values from 0.0 to 1.0, which makes the MAPE score suffer in cases where the ground truth as well as predictions are closer to 0.0.

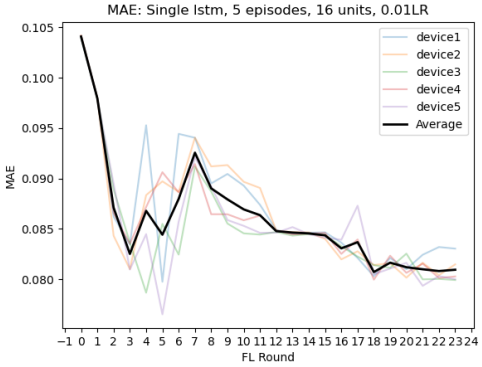
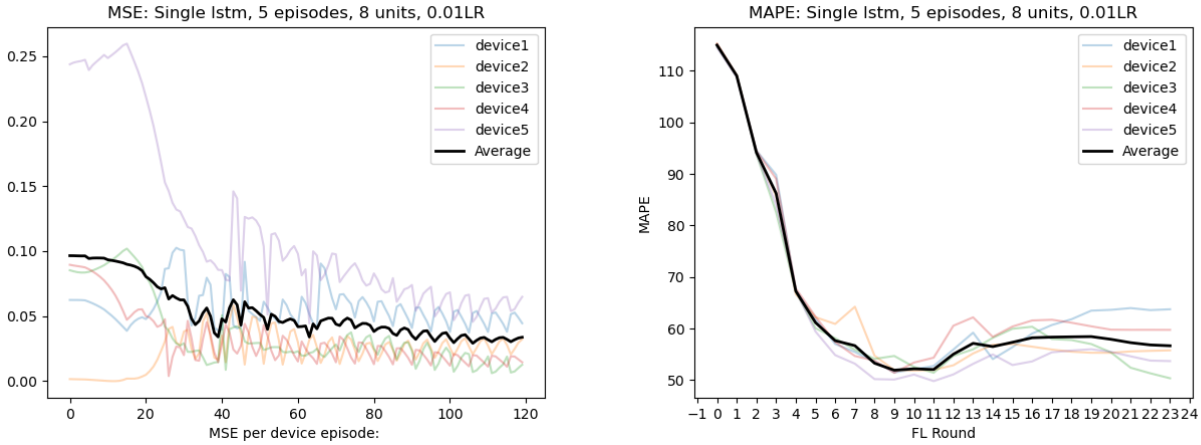


Figure 10: MAE of model in figure 9

To demonstrate the improvement in the model's accuracy overall, the MAE score can be used instead as it does not suffer the same issue of significantly increasing the percentage error when the true value is closer to 0. As such, in Figure 10 it becomes clearer that even if the MAPE shows an increase around the end of the training, this can be due to slightly higher predictions on values with lower ground truths, and not entirely deterioration of the model as a whole due to overfitting. The model is however still objectively quite unbalanced, so finding means to improve it is necessary.



(a) The MSE

(b) The MAPE

Figure 11: 8 Units LSTM. 5 devices training on 500 unique samples with FL.

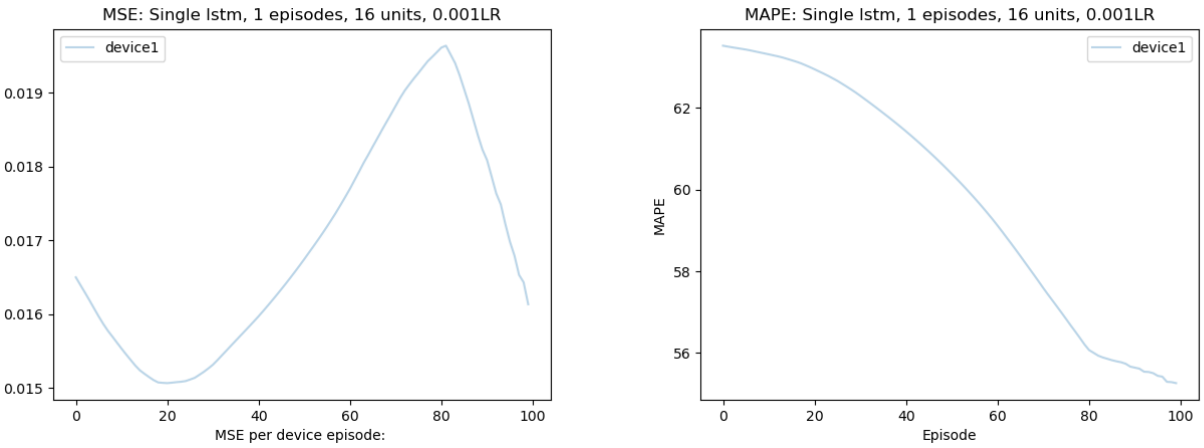
Figure 11 shows a MAPE score which at its lowest point dips below the value of the original and distilled pre-trained model discussed in section 5.2.2. The actual

In order to compare the effect of parameter size figure 11 can be used,

as it has a unit size of 8 instead of 16, meaning it has 28% of the weights of the model in 9. It demonstrates that similar performance curves and even faster converging could be achieved with a smaller model even when trained from scratch. The loss function is more stable Based on the previous rationale for the deviances between the metrics used, this signifies smaller models perhaps having fewer outliers than models with a larger sum of weights. This makes sense as the larger sum of weights will try to learn from more complex patterns of the input data. Based on these measurements it can therefore seem possible that a larger model is more heavily reliant on either a greater dataset or more training on the smaller dataset to decrease the outliers. This raises the question of what the balance of FL rounds and local episodes should be to obtain a better result. Overall though, having a more diverse and greater dataset is generally known to produce better models, as such the relationship between performance when training on a smaller or larger subset of on-device data, the sum of total data in the FL process, and its distribution between devices is also necessary for a more complete picture of performance. These aspects are further discussed in the sections below.

5.4 Federated vs. Centralized Tiny Machine Learning

While the potential benefits from Federated learning have been presented in the form of systematic benefits to privacy, and latency as far less communication is needed, trade-offs can also be addressed as the training of individual models is likely to negatively affect other devices' weights



(a) The MSE

(b) The MAPE

Figure 12: A single device training the same model as in figure 9

Figure 9 demonstrated the potential effect on the loss and metrics when the training only takes place on a single device using a dataset size of 2500 as if it had all 5 devices' data, the learning rate was also decreased as. The loss indicates that the outlier errors are increasing among epoch

5.4.1 Accuracy

The accuracy of the LSTM is assessed by considering the loss function MSE, as well as the MAPE, and MAE metrics, which were tested against the evaluation set. In regards to LSTM, the MAPE reached by the on-device trained model approaches similar values as the original and distilled lstm models highlighted in the compression section 5.2.2. Though training on a smaller dataset, the 16-unit model in figure 9 and its smaller counterpart in figure 11 dip under the 60 MAPE in their best FL round. In general, however, the models are currently incredibly inconsistent, as the regular increase, and thereafter drop in MSE score tells us that there are at least a few significant outliers. It is however important to note that in figure 17 where 2000 data samples are present on each device, ”device2” and ”device3” perform significantly better than ”device3” based on MSE. Considering how previous models have one or more devices using the samples within the range of the same 2000 samples, it then also makes sense why some devices in other presented figures in this section have sudden sharp loss increases as they might be training on flawed data or data that deviates from the norm. Regardless of the high MSE on ”device1” in the aforementioned figure, however, the MAPE on validation is performing better than others. Therefore, based on the comparison of said figure **??** and figure 14 with significantly more weights but the same dataset as the standard model in figure 9, it seems like a greater dataset on-device is more significant than a bigger model with fewer data. Ideally, both concerns would be addressed by having real-time sampling, which is mentioned further in section 6.5.3.

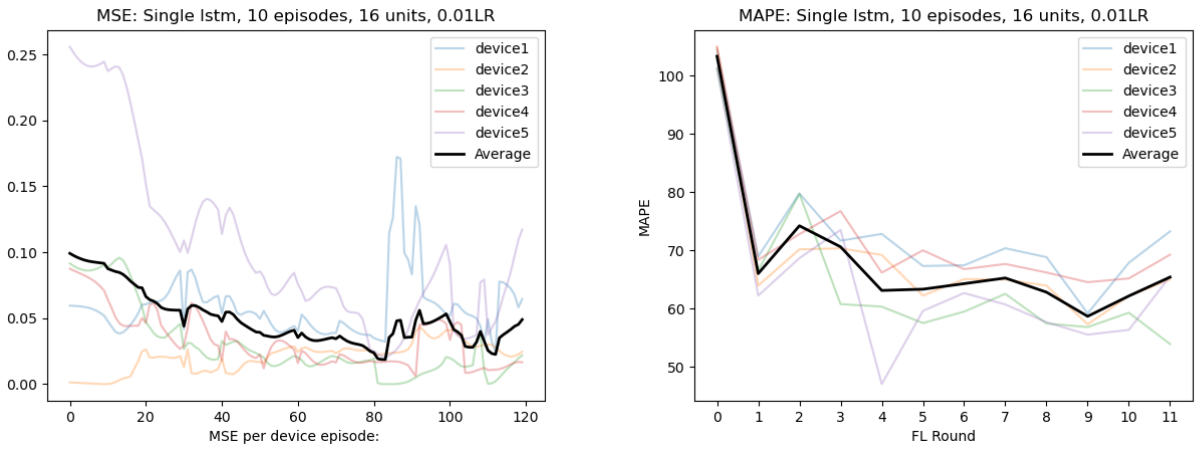
5.4.2 Communication Cost

The communication cost when running FL will naturally increase with the number of dedicated set FL rounds, or with a lower number of local episodes if there is no maximum amount of FL rounds. When attempting to run the LSTM model with half the number of fl rounds and twice the amount of local episodes as seen in figure 13, it resulted in slightly lower peaks of MSE on most of the devices which indicates the higher amount of local episodes enables the individual models to have fewer outliers. The cost of this is however also a slightly higher MAPE. As such the communication cost of federated learning can be cut with such a sacrifice in this particular scenario. An additional means of cutting communication costs during federated averaging is by quantizing the weights from 32-bit floats to 8-bit ints, but this comes at an accuracy loss which can potentially be mitigated by quantization aware training.

5.5 Optimizing the Federated Learning on IoT devices

Some of the strategies attempted to improve the federated learning process when using on-device training. The focus of these tests will be on the training of RNNs using LSTMs in particular.

5.5.1 FL rounds to local episode ratio



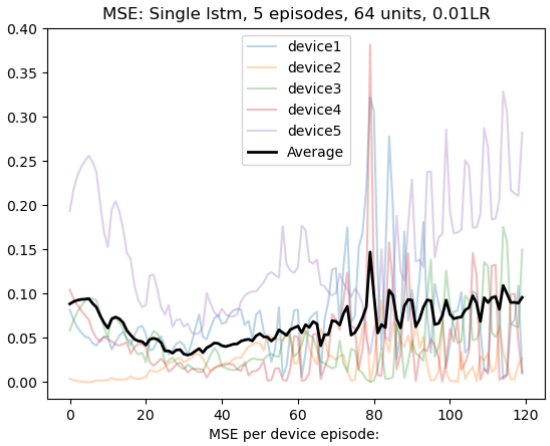
(a) The MSE

(b) The MAPE

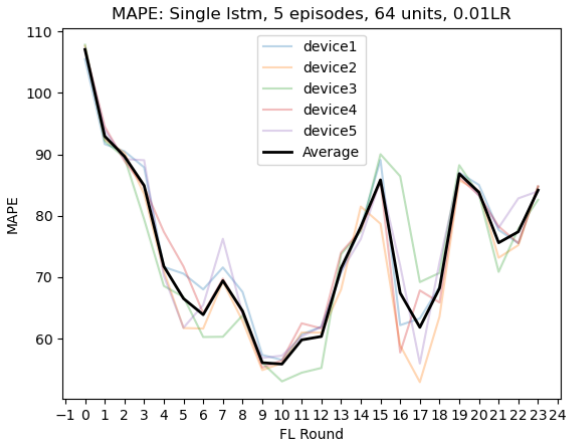
Figure 13: Same model and initial weights as Figure 9, with half the FL rounds, but double the local episodes.

While parameter aspects like weights and data available have known effects on the overall performance of a model, the ratio of local episodes to FL rounds is relevant too as it might affect the time it takes to converge, while allowing the model weights to accommodate the local data well. It would be desirable to reduce the number of FL rounds in order to cut communication costs, but the consequence might be local overfitting as the data they train on is restricted. In the case of figure 13, it is clear that the MAPE suffers in comparison to the counterparts with fewer local episodes.

5.5.2 Model-size to data-set size



(a) The MSE



(b) The MAPE

Figure 14: Same model as Figure 9, but with 64 units instead of 16. Same size dataset-size per device, 500 sequences.

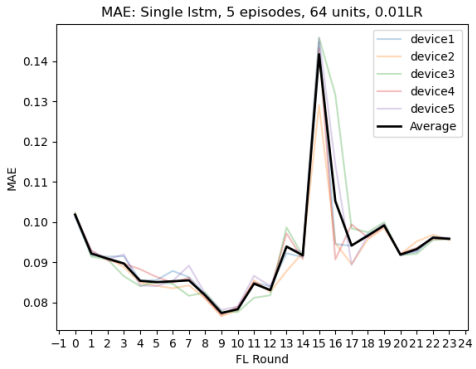
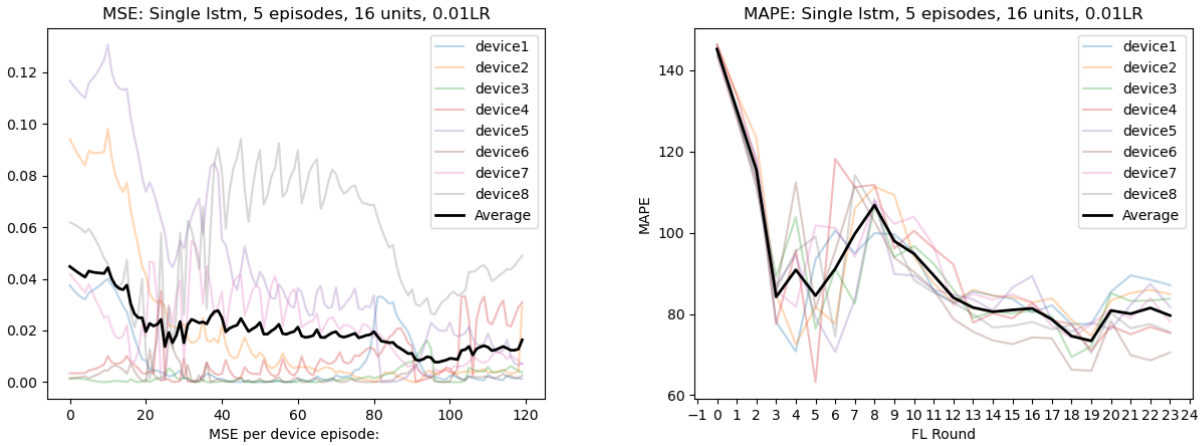


Figure 15: MAE of model in figure 14

As can be seen in the different figures, including 14 and its smaller version 11. Different numbers of weights hold different implications for performance, as the individual models might become overfitted based on the data available, or the training set of some devices align better than others, resulting in an unfair federated learning process with stragglers. The bigger version of these models can be seen in figure 14 and shows this even better in the later stages as ”device5”’s loss function is increasing significantly more than others, while also being a straggler in other, smaller models as well.

5.5.3 More devices with fewer data

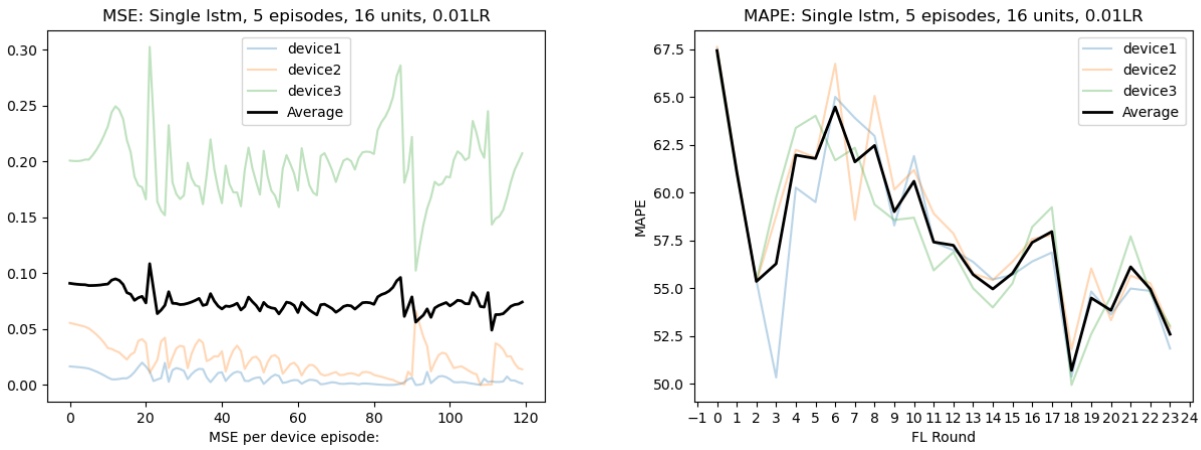


(a) The MSE

(b) The MAPE

Figure 16: Same model as Figure 9, with 3 additional devices and 300 data samples per device

The degree to which the datasets present in total, distributed across all single devices, can be considered part of the whole dataset needs to be addressed. The potential costs to performance by not having all data samples present locally are significant especially in relation to the previously discussed section, because our ability to compensate for lowering the local datasets to afford a greater model size might be worth the cost based on the amount of devices we have to compensate for it. Figure 13 displays the effect of adding 3 devices each with 300 data samples to contribute to the pool of the total dataset. The variability in device performance is a significant issue here as some seem to have questionably low MSE while having a MAPE which conforms to what is expected.



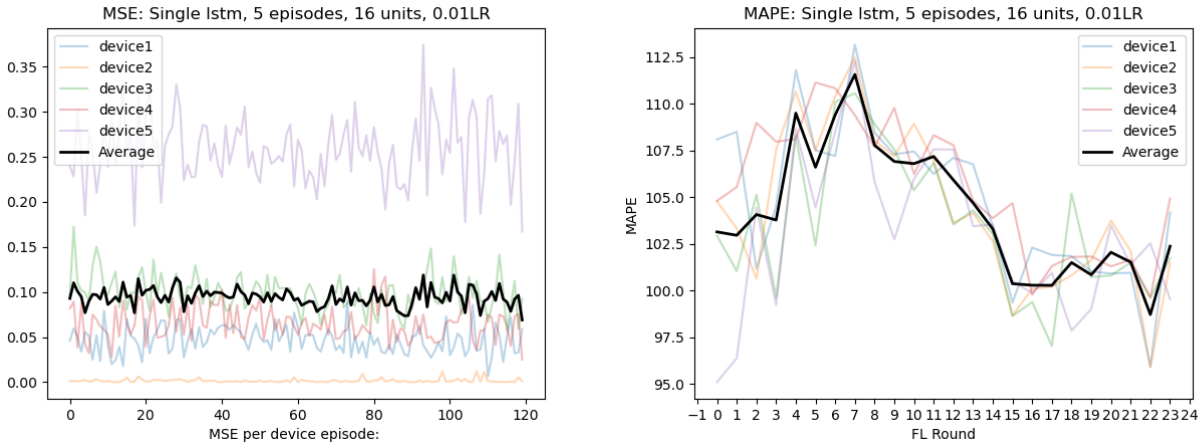
(a) The MSE

(b) The MAPE

Figure 17: Same model as Figure 9, with only 3 devices, and 2000 data samples per device

To contrast the model in figure 16, there is a FL scenario with only 3 devices and 2000 samples of data each can be seen in figure **??**, where the total dataset requires a total of 101.5KB of storage. The effect of having fewer devices with albeit a significantly greater set of data far outperforms the scenario with fewer data and more devices. It is important to consider that each device's validation set is determined based on the size of its training set, so the comparison of the models shows their performance relative to each other with different validation. This would indicate that even if the data is distributed well between more devices, their ability to generalize on the data is significantly lower. The cause for this could be that what little each device gains of knowledge from their smaller dataset ultimately goes to waste as it is not reflected in the performance on the smaller validation set. In addition, when comparing it to the model with a significantly greater number of weights present in figure **??**, it further underlines the issue of on-device generalizations being unusable as especially in a model with a greater number of parameters, it is likely to overfit on the specifics of the limitations. As such, the question turns into questioning what the optimal model size to dataset ratio is, with the number of devices assisting just participating as they cannot compensate for a significant lack of data on other devices unless their models are small enough to automatically obtain generalizations which is applicable to all, at the cost of not having as great of a peak performance. This further highlights desire for datastreaming or live data acquisition.

5.5.4 Quantization Aware Training



(a) The MSE

(b) The MAPE

Figure 18: Same model and initial weights as Figure 9, trained Quantization Aware Training

When compression-aware training takes place, a decrease in performance is expected overall as the weights are trained to suit the loss which is incurred by the compression. In particular quantization-aware training as discussed previously seeks to emulate quantization during training so that the final quantized model is more accurate than a model which has been solely trained on unquantized, standard 32-bit floating point numbers. When testing this on the standard 16-unit lstm model, as seen in figure 18, it is clear that the training process suffers too much from the loss of precision in the conversion. Similarly, as before, one device suffers from severely high loss, and in addition, another device is struggling with what seems like vanishing gradients. Ultimately, deploying quantization-aware training on these smaller models is definitely not a worthy pursuit in this type of implementation. If the model had been larger and trained in a centralized manner then had post-training quantization deployed, in all likelihood the models would have had fair accuracy, but when training the models on devices with limited datasets they struggle with obtaining solid generalizations in general meaning that what little progress is gained more easily disappears in the precision loss of 32-bit float to 8-bit integer quantization during training. It does however raise further questions regarding how effective it would be if dataset could be obtained in real-time, effectively granting the devices a more fair dataset for them to train on as outliers in the dataset would not make devices have such a significant negative effect.

6 Conclusion & Future work

This section concludes the deductions made from my work regarding on-device training of different models with the use of Federated learning. The approach of focusing on LSTM models and time series classification over image classification was due to fact that it is the least explored area within this topic, as such the results evaluated previously and evaluated here will focus on it, though a lot of the general observations can be considered for the other models as well.

6.1 The approach in general

When deploying on device training on a microcontroller, performance was dependent on multiple variables, and if tuned incorrectly they worked against each other more than not, but the performance that could be comparable, albeit on a far lower amount of data, shows potential to be reached with further advancement. Federated learning was seeking to address the privacy concerns regarding centralized data storage, and it effectively solves this issue of global training concern assuming a big enough model is present with enough data. In the context of FL on microcontrollers however the answer isn't as clear as it becomes very dependent on the accuracy needs of the model, and therefore what implications that would have on the size of the model. However, if we can assume that the data can be streamed or sampled in any way, it solves the issue of being concerned about how much data is available, making FL on IoT far more accessible. As such, deploying the LSTM model for predicting PM10.0 on the actual sensors used by NILU, could be a very possible reality.

6.2 Accuracy

The accuracy of the produced models to not compare with the compressed models generated from knowledge distillation, and the limited data I was able to allocate onto the device would negatively impact the pre-trained model as the generalizations learned from the teacher, not the dataset itself, would get replaced by an inferior training process. When considering accuracy for on-device training the most significant aspect to consider is the data allocated more often than not as devices with sufficient data can help each other but devices with too little will obstruct each other's convergence as the wrong generalizations are made. As such, an overall good and permissible performance of on-device trained models, in particular LTSMs, is possible as the variables used in the evaluation indicate a trend of being able to fine-tune the device size to model size as one can start with a smaller model test in relation to the dataset as a smaller model is likely to perform better regardless after the sweet spot is found the new devices with similar dataset size can be added without impacting performance negatively.

6.3 Communication Cost

The communication cost will be relatively high in situations with a low data amount available as a higher amount of local episodes will more likely lead

to overfitting on devices, making the federated averaging detrimental to the overall performance of the devices in general. Assuming that the data is sufficient leaves more room for having fewer FL rounds, but the performance of the other models is supposed to also rely on each other device so ideally there would still be some frequent federated averaging. Quantization, which is used for model compression can also be used for communication, allowing for the transferral of integers instead of floats, as the models would eventually be quantized for inference in the end anyways once FL has been completed, having a model that doesn't suffer much loss for quantization would be ideal. I tried to implement that through Quantization-aware training however the small model, with perhaps the small dataset couldn't learn anything from it, so quantization would have to be applied without any preparation in this case.

6.4 Overall

The approach i chose of relying on data stored on the device proved to be a great hurdle in acquiring new knowledge as being concerned with data distribution and overfitting on the same dataset every FL round became a more significant worry than potential advances that do not concern basic ML problems. In any case, it seems like the current best approach for most accurate predictions microcontrollers is model compression for inference, and facilitating a means to gather sensor data to a more powerful edge node that does the machine learning, then distributing the model for inference would yield better results in most practical applications.

The use of federated learning on IoT nodes is still a significant topic in certain scenarios as cases where the data for training is simple, i.e. from various sensors, should be able to generate sufficiently accurate models in and for the dataset. Such examples include in particular the industrial contexts where various data can be easily gathered and trained on. In addition, deploying ML on such devices would be beneficial instead of an edge node as they might have some actuation power that would be desired to have automated, and latency would place a burden. As well as the fact that when a model is deployed, even if pre-trained it would need to be fine-tuned to the specific scenario to be viable at all.

6.5 Future Work

There are a lot of opportunities for future work in FL for tinyML on-device training based on this project particularly in terms of RNN on microcontrollers. This section will include such future topics which can be used to extend or improve the existing project, and general work that would be a good addition to this work to address issues.

6.5.1 Support more layer types and ML techniques

Increased support for a variety of standard layers, like GRU and transformers for RNNs is necessary to test and compare the different capabilities and costs to better assess which architecture is the most appropriate for different IoT on-device training deployment scenarios. Layers higher in complex-

ity might be favorable as they converge faster and be worth the additional load if it means cutting down on federated learning rounds.

6.5.2 Optimisation for the ML computation on Devices

The current implementation uses dynamically allocated pointer arrays to handle the storage of various arrays of different dimensions and performs the various operations in loops. Solutions leveraging faster and more optimized means for computing these operations would be beneficial in order to speed up the process without incurring much overhead from importing the library. In addition, exploring the different benefits of leveraging device-specific optimizations for different IoT nodes where applicable would make for an interesting comparison of various devices working on the same problem.

6.5.3 Training on Data Gathered Real-Time and Online learning

Because the devices used in this project do not possess the capability of gathering data and generating a data set out of it, conducting a similar experiment with data gathering, especially in the context of RNNs, could allow a significantly higher dataset as old samples could get removed. If implemented, this could allow for in addition to standard training, this enables online learning of various models. As RNN time-series prediction data has the potential to generate its own truths

6.5.4 Alternative FL Algorithms

Algorithms that handle different federated averaging differently to optimize for overall performance would be key in improving my solution as there were always devices with datasets that deviated heavily from the norm. This could include checking for the uniqueness in the weights on each device and assessing whether including them in the update would be detrimental or positive overall. This is especially an interesting topic when using real-time data acquisition as there would always be potential for faulty data to appear.

References

[1] Meisam Booshehri, Abbas Malekpour, and Peter Luksch. *An Improving Method for Loop Unrolling*. 2013. arXiv: 1308.0698 [cs.PL].

[2] Abhradeep Thakurta Brendan McMahan. *Federated Learning with Formal Differential Privacy Guarantees*. 2022. URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm (visited on 05/10/2022).

[3] Han Cai et al. "Once for All: Train One Network and Specialize it for Efficient Deployment". In: *International Conference on Learning Representations*. 2020. URL: https://arxiv.org/pdf/1908.09791.pdf.

[4] Jang Hyun Cho and Bharath Hariharan. "On the efficacy of knowledge distillation". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 4794–4802.

[5] *Complete guide to GDPR compliance*. https://gdpr.eu/. Accessed: 2022-05-25.

[6] Payal Dhar. "The carbon impact of artificial intelligence". In: *Nature Machine Intelligence* 2.8 (2020), pp. 423–425.

[7] "EtinyNet: Extremely Tiny Network for TinyML". In: 36 (June 2022), pp. 4628–4636. DOI: 10.1609/aaai.v36i4.20387. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20387.

[8] Angelo Feraudo et al. "CoLearn: Enabling federated learning in MUD-compliant IoT edge networks". In: *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*. 2020, pp. 25–30.

[9] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[10] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. DOI: 10.48550/ARXIV.1510.00149. URL: https://arxiv.org/abs/1510.00149.

[11] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML].

[13] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[14] Hong Huang et al. *FedTiny: Pruned Federated Learning Towards Specialized Tiny Models*. 2022. arXiv: 2212.01977 [cs.LG].

[15] Kavya Kopparapu and Eric Lin. *TinyFedTL: Federated Transfer Learning on Tiny Devices*. 2021. DOI: 10.48550/ARXIV.2110.01107. URL: https://arxiv.org/abs/2110.01107.

[16] Junhong Lin and Ding-Xuan Zhou. "Online Learning Algorithms Can Converge Comparably Fast as Batch Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.6 (2018), pp. 2367–2378. DOI: 10.1109/TNNLS.2017.2677970.

[17] Nil Llisterri Giménez et al. "On-Device Training of Machine Learning Models on Microcontrollers with Federated Learning". In: *Electronics* 11.4 (2022), p. 573.

[18] H. Brendan McMahan et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data". In: (2016). DOI: 10.48550/ARXIV.1602.05629. URL: https://arxiv.org/abs/1602.05629.

[19] Marc Monfort Grau. "TinyML: From Basic to Advanced Applications". B.S. thesis. Universitat Politècnica de Catalunya, 2021.

[20] *MS Windows NT Kernel Description*. https://luftkvalitet.nilu.no/. Accessed: 2023-04-27.

[21] Thuy T.T. Nguyen and Grenville Armitage. "A survey of techniques for internet traffic classification using machine learning". In: *IEEE Communications Surveys Tutorials* 10.4 (2008), pp. 56–76. DOI: 10.1109/SURV.2008.080406.

[22] Xinchi Qiu et al. "A first look into the carbon footprint of federated learning". In: *arXiv preprint arXiv:2102.07627* (2021).

[23] Leonardo Ravaglia et al. "A TinyML Platform for On-Device Continual Learning With Quantized Latent Replays". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (2021), pp. 789–802. DOI: 10.1109/JETCAS.2021.3121554.

[24] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. "TinyOL: TinyML with Online-Learning on Microcontrollers". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9533927.

[25] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG].

[26] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.

[27] Stanislava Soro. *TinyML for Ubiquitous Edge AI*. 2021. DOI: 10.48550/ARXIV.2102.01255. URL: https://arxiv.org/abs/2102.01255.

[28] Bharath Sudharsan et al. “OTA-TinyML: Over the Air Deployment of TinyML Models and Execution on IoT Devices”. In: *IEEE Internet Computing* 26.3 (2022), pp. 69–78. DOI: 10.1109/MIC.2021.3133552.

[29] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O’Reilly, 2020. ISBN: 9781492052043. URL: https://books.google.no/books?id=sB3mxQEACAAJ.