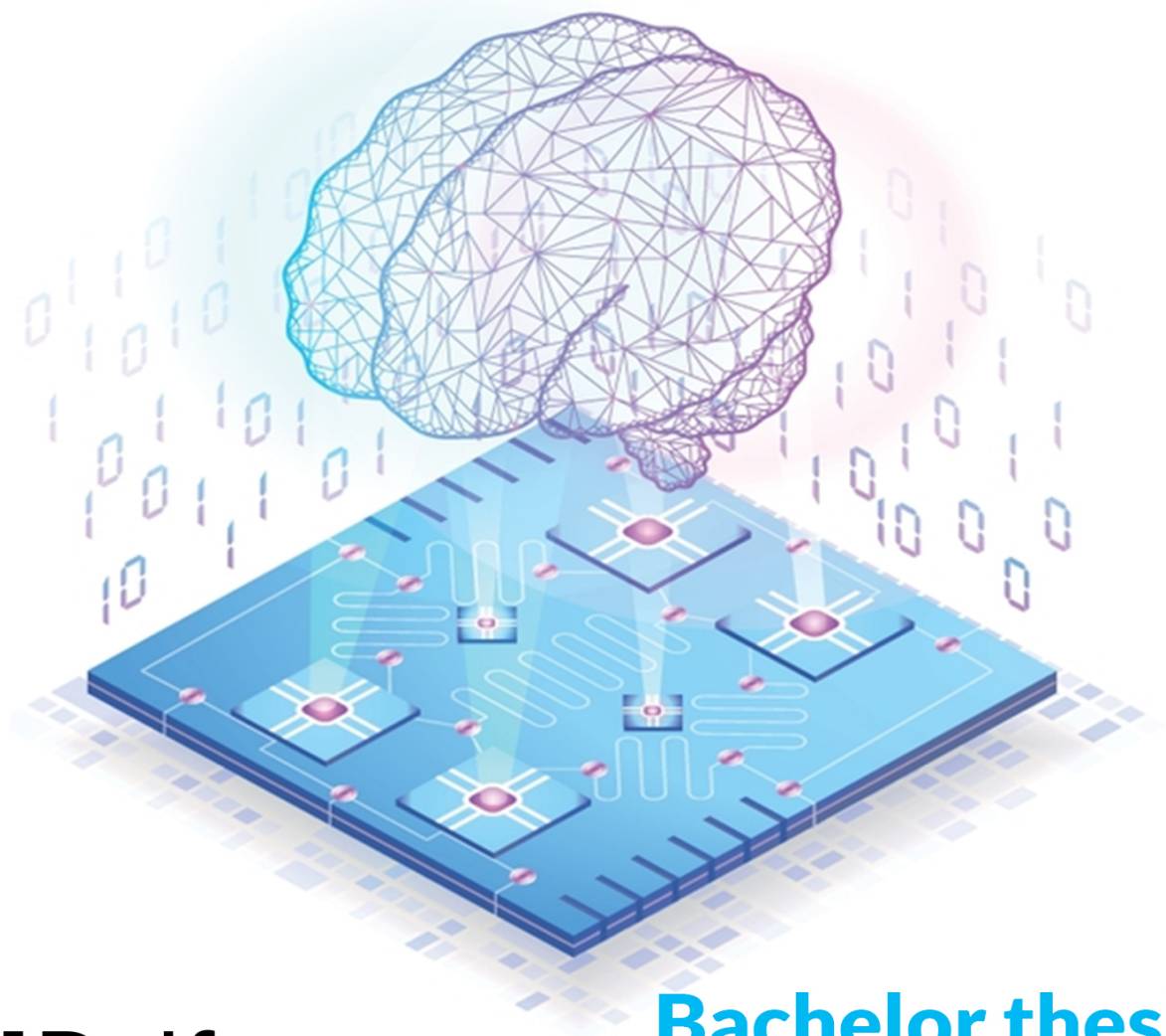
AI ON LOW-COST HARDWARE

Microcontroller Subgroup



TU Delft

Bachelor thesis

BSc Electrical Engineering

JARL BRAND & MANO ROM

AI on Low-Cost Hardware

Microcontroller Subgroup

by

Jarl Brand & Mano Rom

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Wednesday, June 21, 2023, at 4:00 PM.

Students:	J. Y. K. Brand: 5367980 & M. Rom: 5331498	
Project duration:	April 24, 2023 – June 30, 2023	
Thesis committee:	prof. dr. ir. F.P. Widdershoven,	NXP, TU Delft, supervisor
	dr. ir. J. H. G. Dauwels,	TU Delft, supervisor
	dr. C. Frenkel,	TU Delft, supervisor

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

The creation of effective computational models that function within the power limitations of edge devices is an important research problem in the field of Artificial Intelligence (AI). While cutting-edge deep learning algorithms show promising results, they frequently need computing resources that are many orders of magnitude more than the available power and memory budgets for these devices. During the thesis, two unique learning algorithms (backpropagation and forward-forward) were developed and compared using the Teensy 4.1, a low-cost microcontroller board. This work seeks to bridge the gap between the necessary computing efficiency and the hardware's restricted resources.

By creating and analyzing these algorithms, with the Fashion MNIST dataset as a validation set, this thesis creates a baseline for AI efficiency on microcontrollers, with performance targets set at a minimum of 80% test accuracy. The microcontroller software, implemented in C++, is limited to using less than 512 kB RAM for all online training methods. In addition, the potential of transfer learning was also explored.

Key performance parameters, including memory utilization, training and inference times, and accuracy, were analyzed in a comparative study of the backpropagation and forward-forward algorithms. For each learning algorithm, several configurations were explored (such as topologies, and optimizers) to determine the most effective and efficient way for AI implementation on low-cost hardware. The key conclusions of this study reveal that backpropagation demonstrates superior performance in terms of both accuracy and computational efficiency. However, it requires more memory for storing variables, which may be a constraint in on-edge environments. Conversely, the forward-forward algorithm, while achieving lower accuracy, is more memory-efficient, making it a potential choice for less complex tasks or systems with severe RAM limitations.

The application of transfer learning showed potential to accelerate the learning process and to improve the final accuracy, hinting at an effective strategy for deploying advanced AI models on resource-limited edge devices.

Preface

It is with great pleasure that we, Jarl Brand and Mano Rom, present to you this thesis, the culmination of countless hours of effort, learning, and not to mention, a fair share of debugging. This work would not have been possible without the support of several people, whom we would like to express our heartfelt gratitude to.

Firstly, we are also profoundly grateful to our supervisors, prof. dr. ir. F.P. Widdershoven, dr. ir. J. H. G. Dauwels, dr. C. Frenkel, and Y. N. Esparza, for their expert guidance and insights. Their feedback during our weekly meetings was invaluable, and their enthusiasm for the subject was truly contagious.

We would like to extend our deepest thanks to our parents, whose enduring belief in us, constant encouragement, and much-needed supply of caffeine and snacks, served as the fuel that powered this journey. To Jan and Petra, and Adinda and Luc, we couldn't have done this without you.

We would also like to take this opportunity to thank the Technical University of Delft, for the enriching experiences throughout our Bachelor's program. It has been a period of both academic growth and personal development, albeit amidst some rather unexpected global circumstances. The sudden transition to online learning due to the pandemic was not without its challenges. Still, it is safe to say that we have attended more lectures in our pajamas than we ever thought possible. However, the return to campus brought a renewed sense of energy and friendships, and we enjoyed the chance to engage with our peers and professors in person again.

To our fellow students and educators at TU Delft, thank you for making this journey an awesome experience.

Enjoy reading!

Jarl Brand & Mano Rom
Delft, June 2023

Glossary

AI Artificial Intelligence. i, 1–4, 9–11

ANN Artificial Neural Network. 6

ARM Advanced RISC Machines. 2

BP Backpropagation. 24, 27, 28

CNN Convolutional Neural Network. 7, 11

FF Forward-forward learning. 10, 24, 26, 27

FP32 32-bit Floating Point. 10, 12, 14

FPGA Field Programmable Gate Array. 2, 18, 26, 27

FPU Floating Point Unit. 12

IIoT Industrial IoT. 1

KiB Kibibyte. 22

MCU Microcontroller Unit. 2, 12, 17, 26–28

MLP Multilayer Perceptron. 6, 7, 10

RAM Random-access memory. i, 2, 4, 14

ReLU Rectified Linear Unit. 12, 13, 15, 16

SGD Stochastic Gradient Descent. 8, 10, 13–16, 22, 23, 27

TF TensorFlow. 13–15, 25, 26

Contents

1 Introduction

1.1. Background

AI is a rapidly evolving field that is transforming how we perceive and handle data, resulting in advancements in healthcare, finance, transportation, and many more [1]. AI has the potential to improve the efficiency of many systems, but there are limitations to its deployment, particularly in low-cost, low-power devices [2]. AI implementation in such systems frequently necessitates a degree of processing and power resources that exceeds the existing hardware capacities. On-edge AI, designed to handle data locally and in real-time, without relying on cloud-based servers for computational support. This becomes indispensable where connectivity is scarce, latency is critical, or data privacy is paramount [3].

Consider autonomous automobiles, where decisions must be made quickly and reliably based on real-time sensor data. There is no room for lag. Wearable healthcare devices frequently collect sensitive data and must provide real-time health recommendations, making on-edge AI advantageous for both short response times and data privacy. On-edge AI can help with predictive maintenance in industrial IoT (IIoT) environments [4] [5], detecting probable defects or anomalies in machinery. In farming it can process data locally when mobile connectivity is limited [6], saving time and resources. Finally, in smart homes and smart cities, this technology can improve privacy by processing data locally rather than transferring it to the cloud.

However, achieving efficient on-edge AI necessitates the effective implementation of AI algorithms on low-cost, low-power hardware, which is a significant challenge due to the computational intensity of many AI tasks. Consequently, our exploration into the adaptation of AI algorithms, specifically backpropagation and forward-forward learning, for such devices could provide a stepping stone towards more practical and widespread applications.

1.2. Statement of Problem

While AI algorithms are being optimized for high-performance computer systems, less attention has been paid to their implementation on microcontrollers [7]. This discrepancy creates a knowledge gap in efficiently adapting and deploying AI algorithms for resource-constrained edge devices.

As a well-established learning algorithm, backpropagation has been extensively explored in high-performance computing scenarios. Yet, it needs further research for its adaptation to edge devices [8]. On the other hand, the forward-forward learning algorithm, though less utilized in general AI applications, holds potential for low-resource environments, but has not been sufficiently investigated in this context. As such, a thorough understanding of the performance and trade-offs associated with implementing these learning algorithms is lacking.

1.3. Objective of the Study

The primary objective of this study is to implement and evaluate the backpropagation and forward-forward algorithms for on-edge application. This includes understanding the trade-offs involved in memory utilization, training time, inference time, and ultimately, model accuracy. Thus, this study aims to provide a systematic understanding of how best to adapt and optimize these algorithms for resource-constrained edge devices.

To accomplish the objectives of this study, a collaborative approach was adopted, wherein three sub-groups were formed, each consisting of two students. The task of each group was clearly outlined based on their specific roles.

1.4. Sub-group dynamics

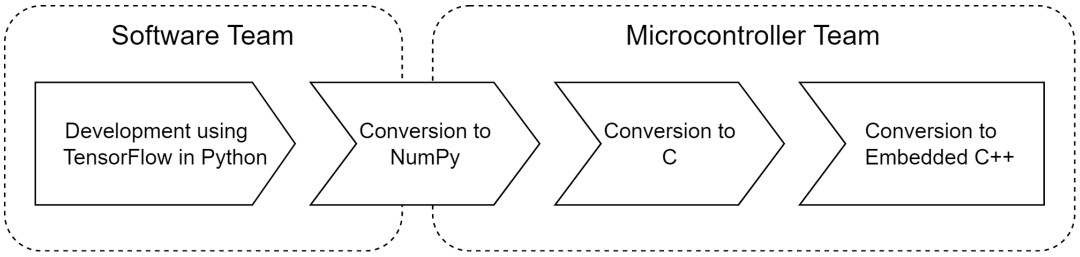


Figure 1.1: Software development pipeline. The software team was primarily responsible for developing the learning algorithms in Python using TensorFlow. The microcontroller team focused on the implementation of these algorithms in C and C++.

The software group was responsible for providing high-level implementations of each algorithm. They utilized the TensorFlow framework in Python, aiming to create a standard against which other implementations could be compared. In close collaboration with the software group, the microcontroller group focused on translating the high-level implementations into a more basic form. To ensure a clear understanding of the underlying processes, they removed the abstraction provided by TensorFlow and instead used NumPy (a mathematics library) in Python for rewriting each algorithm. Following this, the microcontroller group continued by implementing the algorithms on the microcontroller. They wrote a custom linear algebra library and converted the Python model to a C++ version that could be deployed to the microcontroller. This allowed them to test the performance of the algorithms within the specific constraints of a low-cost hardware environment. The software group concurrently performed a hyper-parameter search. The findings were shared with the other groups to ensure uniformity and optimal performance across all implementations. A separate group worked on the implementation of the algorithms on a Field-Programmable Gate Array (FPGA). However, this work was carried out independently of the microcontroller group's work, and little interaction took place between these two groups. The thesis you are currently reading presents the findings of the microcontroller group's implementation and performance evaluations.

1.5. Choice of Hardware and Dataset

The algorithms are implemented on representative low-cost hardware, specifically, the Teensy 4.1 microcontroller board. The choice of this particular board is driven by its capabilities, widespread application, and in-store availability, thereby serving as a relevant case study. Still, the findings are expected to provide broader insights relevant to a range of similar microcontrollers. By using the Fashion MNIST dataset for validation, this study also looks to provide a generic evaluation scenario that could be replicated across various hardware and datasets. To this end, this section presents a review of the Teensy 4.1 hardware, the Fashion MNIST dataset, and why they are appropriate for this study.

The Teensy 4.1, developed by PJRC [9], is an advanced microcontroller board equipped with an IMXRT1062DVJ6 (ARM Cortex M7) MCU and two 512KbRAM chips. Despite some constraints, such as its limited memory and lack of a dedicated AI hardware accelerator, the Teensy 4.1's hardware capabilities coupled with its affordability make it an attractive choice for this study. Its high computational

power relative to its cost makes it a representative case for low-cost AI applications, aligning with the overall goal of our research.

As for the dataset, Fashion MNIST [10] was chosen because of its increased complexity compared to the original MNIST dataset. Composed of 60,000 training and 10,000 test examples, it introduces more variance in the images while maintaining the same format as the original MNIST. Each example is a 28 pixel by 28 pixel, grey-scale image of a piece of Zalando clothing. This dataset provides a more challenging environment for the backpropagation and forward-forward algorithms and yet remains manageable for a microcontroller like the one found on the Teensy 4.1 board. Furthermore, as a well-known dataset, the results obtained from the Fashion MNIST can be easily compared with those of other studies, facilitating the validation of our findings.

More in-depth specifications of the Teensy 4.1 microcontroller board and the Fashion MNIST dataset are provided in appendix A.

1.6. Outline of the Thesis

The thesis opens with chapter 2, 'Program of Requirements,' which outlines the primary constraints and requirements of our study. It sets the foundation for our research by defining the problem, the objectives, and the criteria for the assessment of the learning algorithms.

Following this, chapter 3, 'Background and Related Works', provides a comprehensive review of the existing literature and studies in this field. This chapter helps situate the thesis within the broader scientific context.

Chapter 4, 'Methodology' then details the specific steps taken to implement and evaluate the backpropagation and forward-forward algorithms. It covers the intricate techniques employed, the optimizations made, and the experimental setups that were used to collect the results.

Chapter 5, 'Results and Discussion' presents our findings and engages in a thorough analysis of the data. It compares the performance of the two algorithms and discusses the implications of these results in terms of memory usage, training time, inference time, and accuracy.

Finally, in chapter 6, 'Conclusion', we reflect on our research, summarizing the key findings and discussing their implications.

2

Program of Requirements

The aim of this project is to develop and implement selected AI algorithms efficiently on a low-cost microcontroller, in the context of on-edge application. The selection of algorithms and defining their respective boundary conditions form the core of this project. This Program of Requirements provides a comprehensive summary of the implementation sequence of the chosen algorithms and the restrictions brought about by hardware limitations.

2.1. Functional Requirements

The core functional requirement of the project is to enable the efficient running of the chosen AI algorithms on a low-cost microcontroller. These algorithms are:

- Backpropagation
- Forward-Forward algorithm

The successful implementation and evaluation of each algorithm will follow a stepwise process, commencing with a simple base case and gradually extending to the hardware's limits:

1. Offline single-layer implementation using TensorFlow and TensorFlow Lite (guidance from Software group)
2. Offline single-layer implementation using Python and Numpy
3. Online single-layer implementation using C++
4. Offline multi-layer implementation using TensorFlow and TensorFlow Lite
5. Offline multi-layer implementation using Python and Numpy
6. Online multi-layer implementation using C++
7. Transfer learning using a convolutional network

2.2. Mandatory Requirements

The mandatory requirements for all training algorithms are:

- The software for the microcontroller must be written in C or C++.
- All online training algorithms must use less than 512 kB RAM.
- Test accuracy after full training on Fashion MNIST should be over 80%.
- Accuracy on-device learning should be nearly identical to full precision TensorFlow models (within 2%).
- Implement Backpropagation and Forward-Forward on the microcontroller.

2.3. Trade-Off Requirements

The trade-off requirements listed below are open for consideration and adjustment to achieve an optimal balance between various desired outcomes. Each requirement includes a specified target. Although improvements beyond these targets are desirable, they should not supersede the optimization of other trade-off requirements.

- Minimize the memory use of all algorithms, striving for a target lower limit of 256 kB RAM - a common memory size for microcontrollers.
- Minimize the time required for training, aiming to achieve a goal of less than 1 minute per epoch.
- Minimize inference time, targeting a maximal duration of 0.5 seconds per batch of 16 images.
- Maximize the accuracy of trained networks on the Fashion MNIST test dataset. This requirement does not have a specified target but should always be optimized.
- Experimenting with different topologies, with a minimum of online single-layer.

3

Background and Related works

This chapter delves into the specifics of the backpropagation and forward-forward learning algorithms, which are the primary focus of our study. By comprehensively reviewing the existing research related to these areas, we aim to set a robust foundation for the forthcoming investigation into the performance of these algorithms on the microcontroller.

3.1. Neural Networks

Artificial neural networks [11] (ANNs) consist of multiple layers of interconnected artificial neurons, or nodes. Each node is connected to a set of other nodes, and is triggered by an activation function, which determines whether the neuron should be activated based on the weighted sum of its inputs. However, despite the neurons' apparent simplicity, neural networks can learn complex behaviors.

3.1.1. Multilayer Perceptron

The Multilayer Perceptron [12] (MLP) is a type of feedforward artificial neural network, often forming the foundation for basic implementations on hardware. An MLP is characterized by one or more layers of perceptrons, with each perceptron formulated as:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{3.1}$$

Where:

- y is the output of the neuron.
- f is the activation function.
- w_i are the weights associated with the inputs.
- x_i are the input values.
- b is the bias term, providing an additional degree of freedom.
- n is the number of inputs to the perceptron.

The MLP is typically trained by backpropagating an error gradient and then using an optimizer which aims to minimize the discrepancy between the network's output and the actual target output [12] [13].

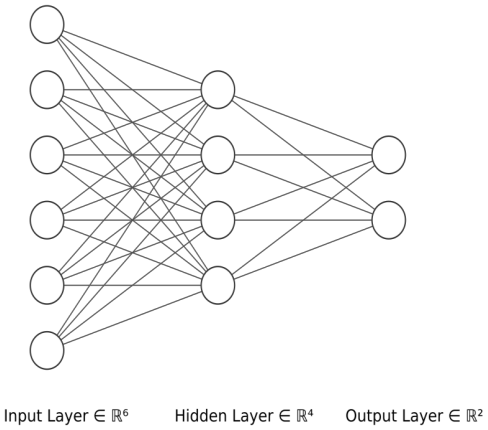


Figure 3.1: An example of a Multilayer Perceptron network

3.1.2. Convolutional Neural Network

Convolutional Neural Networks [14] [15] (CNNs) are a class of deep learning models most commonly applied to analyzing visual data. CNNs are designed to automatically and adaptively learn spatial hierarchies of features from tasks such as image classification.

CNNs are organized in three dimensions (width, height, and depth), as can be seen in fig. 3.2, and use convolutional layers, pooling layers, and fully connected layers for the extraction of features from input images. After feature extraction, the fully connected layers utilize these features for classification.

The network is typically trained by backpropagating an error gradient and then using an optimizer, to adjust weights and biases based on this gradient.

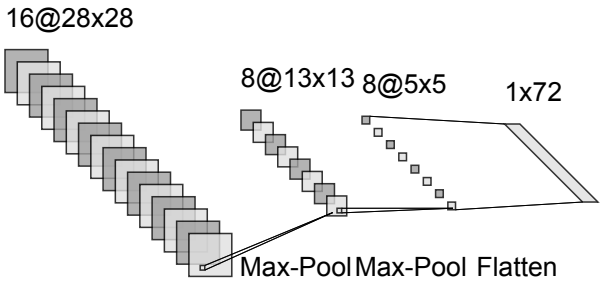


Figure 3.2: An example of a CNN with two hidden layers

3.2. Backpropagation

Backpropagation [13] is the cornerstone of learning in most modern neural networks and an essential part of the training process for multi-layered networks, including MLPs and CNNs. The backpropagation training process can be divided into two phases: forward propagation and backward propagation.

3.2.1. Fundamentals of Backpropagation

During forward propagation, the input is passed through the network, layer by layer, until an output is generated. This output is then compared to the expected output using a cost function (also known as a loss or objective function) to determine the error of the prediction.

In the backward propagation phase, the error computed is propagated backward through the network. This is where the 'backpropagation' algorithm gets its name. The error is used to calculate the gradient of the cost function with respect to the weights in the network.

The weights are then adjusted in the direction that minimizes the cost function. This is done using

the method of gradient descent or one of its variants (such as stochastic gradient descent, mini-batch gradient descent, or others). The rate at which the weights are adjusted is controlled by a parameter called the learning rate.

Derivation of Backpropagation
The key mathematical insight of backpropagation is the application of the chain rule to compute the gradient of the cost function with respect to the weights and biases. Given a cost function $C(w, b)$ that depends on the weights w and biases b of the network, the chain rule yields the following:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}} \tag{3.2}$$

Here a_j is the activation of the j-th neuron in the next layer, and z_j is the weighted input to that neuron. The terms $\frac{\partial C}{\partial a_j}$, $\frac{\partial a_j}{\partial z_j}$, and $\frac{\partial z_j}{\partial w_{ij}}$ can be computed relatively easily, making it feasible to compute the gradient $\frac{\partial C}{\partial w_{ij}}$ [16].

3.2.2. Stochastic Gradient Descent
Stochastic Gradient Descent [17] [18] (SGD) is a variant of the traditional Gradient Descent algorithm, which is used to find the local minimum of a function. SGD introduces an element of randomness into the optimization process, which can help to avoid getting stuck in local minima and speed up the computation.

Basic Principle of SGD
The basic principle of SGD is to estimate the gradient of the cost function by calculating it for a small randomly-selected subset of the data, rather than for the entire data set as in standard Gradient Descent. This subset is referred to as a mini-batch. For each mini-batch, we calculate the gradient and update the model's parameters (e.g., the weights in a neural network) in the direction that minimizes the cost function. This process is repeated until the algorithm converges to a minimum.

Advantages and Disadvantages
The main advantage of SGD is its efficiency. Because it uses only a small subset of the data at each step, it is much faster than traditional Gradient Descent, especially for large datasets, which is particularly important in memory-constrained microcontrollers [19].

SGD also has some drawbacks. Its convergence is less stable than traditional Gradient Descent, the cost function does not decrease smoothly but fluctuates up and down, although the general trend is downwards. This fluctuation can be reduced by decreasing the learning rate over time. Another challenge is the selection of an appropriate learning rate and mini-batch size, which can significantly impact the efficiency and effectiveness of the learning process.

3.2.3. Learning Rate Decay
The learning rate in the context of neural networks is a hyperparameter that determines the step size at each iteration while moving toward a minimum of a loss function. It greatly influences how quickly or slowly a network learns during the training phase. However, setting an optimal learning rate can be challenging. If it is set too high, the loss function may oscillate or diverge; if it's too low, convergence may be slow or the function may get stuck in a local minimum.

Learning rate decay [16] is a common strategy to tackle this issue. The idea is to start with a relatively high learning rate to benefit from fast initial learning, and then gradually reduce it over time. This allows the model to make large updates to the weights early on when the errors are usually high, and smaller updates later when trying to fine-tune the weights.

3.2.4. Momentum
Momentum is another technique used to accelerate the training of a neural network and to mitigate the risk of getting stuck in local minima. The concept of momentum is borrowed from physics: a ball rolling

downhill will keep its momentum and continue to roll even when it reaches a shallow or flat area.

In the context of neural networks, when we update the weights with gradient descent, we also take into account the previous change in weights. We add a fraction of the update vector of the past time step to the current update vector. The addition of momentum to the weight update rule causes past gradients to influence the current direction in the weight space.

Mathematically, the weight update rule with momentum can be represented as follows:

$$V_t = \mu \cdot V_{t-1} + (1 - \mu) \cdot \eta \cdot \nabla_w C \tag{3.3}$$

$$w = w - V_t \tag{3.4}$$

Here:

- V_t is the velocity (the amount by which the weights are adjusted) at time step t.
- μ is the momentum term, which is a hyperparameter between 0 and 1.
- V_{t-1} is the velocity at the previous time step.
- η is the learning rate.
- $\nabla_w C$ is the gradient of the cost function with respect to the weights.

The use of momentum helps the model navigate along the relevant directions and diminishes oscillations, thus leading to faster convergence [16].

3.2.5. Strengths and Weaknesses of Backpropagation

Backpropagation offers several advantages. It is generalizable and can be applied to any differentiable system. It also has strong empirical success across a wide range of tasks, including those involving large and complex data.

However, it also has its drawbacks. Backpropagation requires substantial computational resources, which can make the networks slow on a microcontroller, and requires not only the weight matrices and bias vectors to be stored but also their gradients and intermediate values calculated during the forward pass, which makes it challenging to implement on low memory hardware such as microcontrollers. Moreover, it requires complex hyperparameter tuning to ensure convergence and avoid local minima during optimization.

3.2.6. Applicability to this Thesis

Numerous studies [20] [21] have proposed solutions to optimize the implementation of the Backpropagation algorithm on resource-constrained hardware. These solutions typically involve trade-offs between accuracy and computational requirements. Given the large body of research and discoveries made in the domain of Backpropagation on a microcontroller, this will serve as a reference to other algorithms to be implemented on the microcontroller.

3.3. Forward-forward

This section covers the forward-forward algorithm [22], another learning algorithm in AI, with potential benefits for on-edge applications.

3.3.1. Overview of the Forward-Forward Algorithm

The Forward-Forward algorithm operates on the principle of updating the weights of the network layers sequentially from input to output, hence the name 'forward-forward'. When applied in the context of classification, the input and a label are given to the network, the networks goal is to maximize the output when a correct label is given and minimize the output for an incorrect label. Unlike backpropagation, which computes the error signal at the output and propagates it backwards, the forward-forward algorithm bases the weight updates on the neuron outputs in the previous layer.

3.3.2. Fundamentals of Forward-Forward

The forward-forward (FF) algorithm uses an incremental learning approach, where the network's weights are updated iteratively based on the neuron's outputs from the previous layer.

The input layer consists of the flattened input data and a label, the network will either get positive data (data with the correct label) or negative data (data with an incorrect label). The hidden layers are fully connected MLPs, the networks output p is the probability that the data is positive, given by applying the logistic function σ to the summed squared output for each neuron y in that layer j minus a threshold θ, making it the neural networks objective to maximize this sum when positive data is applied and minimize the sum when negative data is applied, this is shown in eq. (3.5) as presented in the original paper by Hinton [22].

$$p = \sigma(\sum_{j} y_j^2 - \theta) \tag{3.5}$$

In the training phase, the weights can be updated by using SGD as described in section 3.2.2, on every layer individually instead of across the whole network. The weight update rule highlights one of the primary distinctions of the forward-forward algorithm, that is, the error used to update a weight depends only on the target and output of the current layer, and the output of the previous layer. This iterative procedure is repeated for each layer in the network, from the first hidden layer to the output layer, thus completing one epoch of training.

3.3.3. Advantages and Disadvantages

While the forward-forward algorithm's mathematical simplicity is appealing, it's worth noting that its learning dynamics can be substantially different from those of backpropagation, as will be further explored in chapter 4 and chapter 5. The primary advantage of the forward-forward algorithm is its locality of information during the weight update process, which leads to less memory utilization.

However, one potential drawback is that the forward-forward algorithm may be slower to converge or may get stuck in poor local minima due to the lack of global error information. Another drawback is that during inference, each image is tested against every possible label, where the highest score is the networks prediction. This increases the amount of forward passes required for each classification.

3.4. AI in microcontrollers

This section provides an overview of the current state of AI implementation in microcontrollers. It explores the unique challenges and opportunities presented by these low-cost, low-power devices and reviews the techniques and approaches that have been proposed to overcome the computational and power constraints in these systems.

3.4.1. Offline Learning

Offline learning refers to the training of a neural network on a high-performance computing machine, followed by its deployment on an edge device for inference. The typical practice during model training is the utilization of 32-bit floating-point (FP32) precision. The weights of the network are then scaled down to integers through a process known as quantization. This process not only minimizes the storage requirements of the model but also permits the usage of more compact hardware architectures. Previous studies have established that this quantization process can be performed without incurring significant accuracy losses [23]. TensorFlow Lite, in particular, provides an assortment of tools designed to facilitate the quantization of neural networks.

3.4.2. Online Learning

The concept of online learning involves performing (a portion of) model training directly on the edge device. This approach provides the advantage of enabling the model to adapt to changes in the environment. Backpropagation coupled with a variant of Gradient Descent is the standard approach for model training. However, these methods are memory-intensive, making them unsuitable for devices

with limited memory capacity. Moreover, models are typically quantized, which poses additional challenges for optimization [20].

3.4.3. Transfer Learning

Transfer Learning, a technique that combines aspects of both online and offline learning, is sometimes used as a solution. The initial phase of this approach mirrors offline learning, wherein a neural network is trained on powerful hardware and then the weights of this network are frozen and quantized, preventing any further modifications on the microcontroller. Following this, an additional layer is added to the network, which will be (re)trained on the edge device. Whenever a label is available for input data, the weights of the final layer are updated using an online gradient descent algorithm [24].
In this configuration, the offline layers will do feature extraction and the online layers will do classification. This can be particularly advantageous when the model extracts features of a moving dataset, wherein in case the data changes over time, it can detect new patterns.

One of the major benefits of using CNNs in the context of transfer learning lies in their ability to efficiently extract features from input data. The initial layers of a CNN, trained on a large-scale dataset, can identify and extract generic features like edges, curves, and textures. These features often have universal utility and can be applied to a variety of different tasks. Thus, these pre-trained CNN layers can be used as a fixed feature extractor while the subsequent layers of the network can be trained on a specific task.

The application of transfer learning with CNNs not only saves computational resources but also time, as the network does not need to learn these features from scratch. The pre-trained CNN can be quantized, significantly decreasing memory utilization, and a smaller network can then be trained online to adapt to specific tasks or environmental changes.

A possible use case for this could be a wearable device for health monitoring. A CNN pre-trained on a large database of biomedical signals could be used to extract relevant features from incoming sensor data, and a smaller network could be trained online to learn the specific patterns of the individual user. This could be used to monitor and alert about changes in the health status of the user, personalized to their unique physiology.

3.4.4. TensorFlow and TensorFlow Lite

TensorFlow [25], developed by Google Brain Team, is one of the most popular open-source libraries for machine learning and neural networks. It provides a comprehensive ecosystem of tools, libraries, and resources that facilitate building and deploying machine learning applications. It's well-suited for large-scale machine learning tasks with high computational demands, providing support for distributed computing.

However, when considering deployment on edge devices, the full TensorFlow library can be excessive due to its high computational and storage requirements. This is where TensorFlow Lite comes into play. TensorFlow Lite is a streamlined version of TensorFlow designed specifically for deployment on mobile and edge devices. It allows models to run on resource-constrained devices, focusing on efficiency and performance.

In the context of this study, TensorFlow Lite is particularly relevant as we aim to implement AI solutions on the low-cost, low-power Teensy 4.1 microcontroller. TensorFlow Lite provides the tools necessary to train models offline and implement the quantized models on this hardware.

4

Methodology

4.1. Algorithm Implementation and Optimization

This thesis explores the implementation of the backpropagation and forward-forward algorithms on the microcontroller to train neural networks. First, considerations regarding each training algorithm are presented. Finally, transfer learning is implemented using both algorithms. Information about the support structures for these algorithms (data loading, linear algebra library, etc.) can be found in appendix C. Care is taken to statically allocate contiguous memory, instead of using dynamic allocation. This can be faster on microcontrollers and avoids the problem of memory fragmentation [26].

Full precision (FP32) is used for the implementation of learning algorithms on the microcontroller. This allows for a direct comparison of the results to the models trained with TensorFlow. Also, if the gradient is 8-bit integer quantized, the gradient may not be accurate enough to converge properly [27]. There are papers that have addressed the issue[27][20], but within the scope of this thesis it is not possible to replicate these works. By using full precision for all learning algorithms on the MCU, a fair comparison can be made regarding their performance. Furthermore, the Teensy has an FPU, which allows for floating point math at about the same speed as integer math[9].

4.1.1. Backpropagation

The backpropagation algorithm is evaluated in both an offline and online setting. First, a model is trained in Python using TensorFlow on a computer, after which the trained model is inferred on the microcontroller. The same models are also trained on-device, such that performance can be compared.

Network Topologies & Hyperparameters

Two main neural network topologies are used to evaluate the backpropagation algorithm. The images in the Fashion MNIST datasets are 28x28 grayscale. These images are flattened to form an input dimension of 784. There are 10 classes, so the output dimension is 10. First, a network with a single hidden layer is trained. This layer has 32 neurons. This model shall serve as a baseline implementation to compare other models against.

A network with two hidden layers is also trained, with both layers consisting of 32 neurons. The amount of neurons is mainly chosen because it easily fits in memory and inference and training times are low, making it easier to test multiple configurations of hyperparameters. Apart from this, it is arbitrary.

For all layers except the output layer, the ReLU activation function is used. For the output layer, softmax is used to obtain class probabilities. The loss is sparse categorical cross-entropy. Its implementation and derived gradient are adapted from [28]. The network topologies are shown in fig. 4.1.

For both training processes, the same set of hyperparameters is used. Stochastic Gradient Descent is used as an optimizer. Learning rate decay and momentum are implemented in both training processes.

Most of the hyperparameter search is carried out by the software team, as they are able to iterate faster using TensorFlow. The hyperparameters that are used are as follows:

- Learning rate: 0.1 with learning rate decay, 0.01 without learning rate decay.
- Learning rate decay: 0.95 every 200 batches with a lower limit of 0.01 (when applied).
- Momentum: 0.9 (when applied).
- Batch size: 16: this results in lower memory usage and faster convergence than higher batch sizes.

Two of the hyperparameters (learning rate of 0.01 and momentum of 0.9) are chosen because they are the default values for SGD in TensorFlow and are thus expected to provide reasonable results.

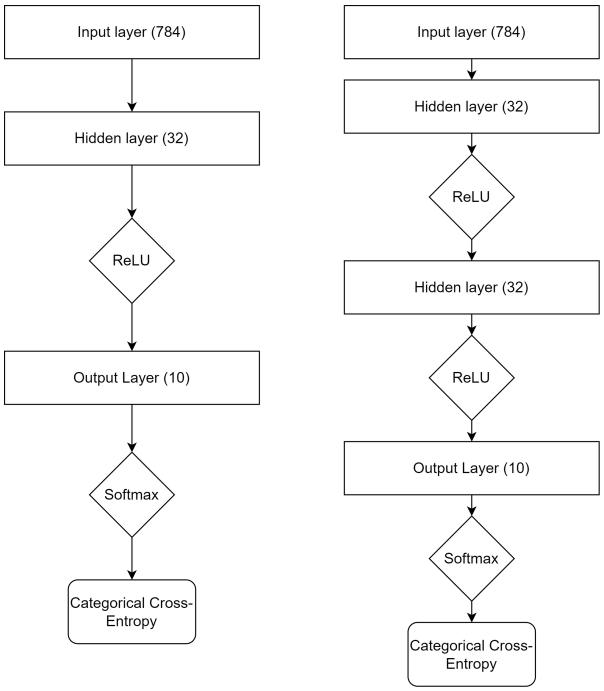


Figure 4.1: Network topologies of the networks that are trained using backpropagation. Layer dimensions are shown in parentheses. The network with a single hidden layer (baseline) is shown on the left and the network on the right has two hidden layers. Both networks use the ReLU activation function for the hidden layers and the softmax activation function for the output.

Offline learning & TF Lite inference

In order to run inference-only models, a way is needed to run the models that are trained on the computer using TensorFlow on the MCU. TensorFlow Lite Micro provides such functionality, as described in chapter 3. In appendix C it is described how it was ported for use on the Teensy.

TensorFlow Lite Micro is able to infer TensorFlow Lite models. The models are first trained on a computer using Python and TensorFlow by the software team. Then, they are converted into a TensorFlow Lite model. During this conversion, post-training quantization is applied, in which the network weights are converted from 32-bit floating point values to 8-bit fixed point integer values. TensorFlow Lite also calculates appropriate zero points and scaling factors for the input and output tensors. These are required to quantize and dequantize the input and output of the model respectively. 8-bit integer quantization is chosen over 32-bit floats for inference, because the accuracy often does not suffer significantly [29] and due to the 4x smaller memory footprint, it is popular for model inference on microcontrollers.

The conversion process produces a `.tflite` flatbuffer file. This file is then converted into a c++ file, including the flatbuffer in an array. A TF Lite model interpreter is initialized with only the operations

that the model uses. This saves memory when compared to initializing the interpreter with all operations enabled. Furthermore, memory is allocated for the interpreter to carry out its calculations.

Inference of each model is accomplished by loading in one example at a time, quantizing said example using the previously found zero point and scaling factor, and invoking the model interpreter. After the model has been inferred, the output is dequantized.

Online learning
In order to facilitate online learning, the backpropagation algorithm is implemented in Embedded C++. For this to be successful, the forward and backward pass are first worked out manually, following the software development pipeline outlined in chapter 1.

Once these are worked out, they are written in C++ using the custom linear algebra library (appendix C) to perform all necessary calculations.

All required matrices and vectors for the weights, biases, gradients, and intermediate values are statically allocated. First, the weight matrices and bias vectors are initialized according to a normal distribution and scaled down. The scaling is important because this prevents that the network outputs high confidence scores in the beginning, which results in slower training in the beginning. Instead, at the start of training, a uniform distribution over all classes is expected at the output, which is more the case after scaling. The scale factor may also not be too small, otherwise, the gradients will also become small and the model will train slowly in the beginning.

After initialization, training is started. First, a batch of training examples is loaded from the SD card. The rows of the input matrix represent the batch dimension and the columns the feature (pixel values in this case) dimension. Then, the forward pass is executed. For each layer, the input matrix for that layer is multiplied by its weight matrix. Then, the bias vector of the layer is added to the resulting matrix as a row vector. Finally, an activation function is applied to the resulting matrix. The loss over the batch is calculated by applying categorical cross-entropy on the output of the network. After the forward pass, the backward pass is executed. In this stage, the gradient of the loss with respect to the weights and biases is calculated. Several optimizations are introduced to enhance the learning process and speed.

First of all, the calculations in the backward pass rely on multiplying the transpose of a matrix with another matrix. These two operations are combined into one in order to save memory. Additionally, there is a memory-optimization possible when the optimization algorithm is regular SGD. It is possible to directly update the weights, instead of first calculating a gradient matrix and then updating the weights. This approach can save a large amount of RAM. For example, if a weight matrix is of shape 784x32 (784 is the image input size, and 32 the number of neurons), the gradient matrix must be of the same size. If the gradients are FP32, the amount of memory that is saved with this approach would be $784 \cdot 32 \cdot 4 = 98$ kB. Making this optimization does, however, mean that momentum based optimizers cannot be used, as they rely on keeping a moving average of the gradient matrices in memory. Because TensorFlow does not make this optimization, it is not used when directly comparing backpropagation on the microcontroller with backpropagation in TensorFlow. A similar optimization is applied to momentum: the moving average is directly applied inside the gradient matrix.
In case the optimization is not enabled, an extra step is required to apply SGD to the network weights and biases. During this step, momentum is also calculated and applied. Furthermore, in both cases, the number of processed batches are tracked in order to apply learning rate decay.

4.1.2. Forward-Forward

In contrast to backpropagation, the forward-forward algorithm performs learning locally (with local meaning one layer at a time). Furthermore, it cannot be made to output class probabilities in the final layer. These characteristics of the algorithm make it so that both training and inference are different from what is the case for backpropagation. In this section, the implementation of both inference and training is presented (offline using TF Lite Micro and online).

Network Topologies & Hyperparameters
As mentioned earlier, when training using the forward-forward algorithm, there is no output layer that explicitly outputs class probabilities. This has consequences for the network topology. Two network topologies are presented in fig. 4.2. The first network has a single layer of 32 neurons. The second network has two layers of 32 neurons each. The choice to make the second layer have 32 neurons is influenced by the implementations [30], including the original paper [22] that all feature layers of the same dimensions. This also allows for a more direct comparison with backpropagation. The activation function for all layers is ReLU. In between layers, layer normalization is applied. This is done in order to make the second layer learn new features [22].
The loss function is defined per layer. The loss function is contrastive and is equivalent to [30]. Its mathematical definition and gradient derivation are shown in appendix D. It is designed to pull the neuron activations above a certain threshold (which is a hyperparameter) for ”positive” data and below the threshold for ”negative” data.

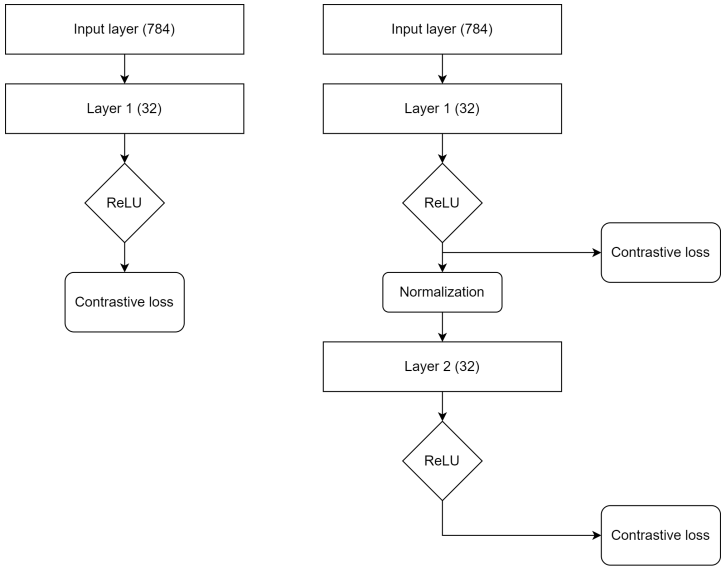


Figure 4.2: Topologies of the networks trained with the forward-forward algorithm. Layer dimensions are shown in parentheses. The activation function is ReLU for all layers. Layer normalization is applied in between layers. The loss is calculated over the layer activations before normalization

The optimizer that is used is SGD without momentum and learning decay. The learning rate for the first layer is 0.1 and for later layers it is 10. This is done because during the testing phase using TensorFlow and NumPy, it was found that the later layers train much slower than the first. The loss threshold is set to 2. A batch size of 16 is used. This is the same as for backpropagation and allows for direct comparison of memory usage.

Offline learning & TF Lite Inference
Using TensorFlow, the two layers are trained and quantized seperately by the software team. On the microcontroller, TF Lite Micro is used to infer layers seperate from each other. This is required, because it is not possible to extract the independent layer activations from TF Lite Micro after inferring multiple layers at once. The normalization in between the layers is not done by TF Lite Micro. Predictions are generated following the approach outlined in chapter 3.

Online Learning
Before any data is put through the model, the input data has to be made positive or negative. There are multiple ways of doing this. One could pre-process the training dataset and make two copies of it: a positive dataset and a negative dataset. It is also possible to generate positive and negative data on the fly. In this approach, first a batch of data without encodings is loaded into memory along with the corresponding labels. Then, the data is made positive by encoding the true labels. After encoding, a training step is run on this data. Then, the positive encodings are overwritten by negative encodings

and the training step is repeated. This method has the benefit of not requiring any pre-processing, which may be more representative of a use case in which the data is an incoming stream instead of a static dataset. The drawback is that the network is required to train on the same batch each time for both positive and negative data. This approach is chosen because of the aforementioned benefits.

At the start of training, the weight matrices are initialized randomly. After this, the forward pass is executed up until the layer currently under training. If the layer in question is not the first layer, layer normalization is applied. After this, the forward pass functions like any other forward pass, with matrix multiplication, row vector addition, and a ReLU activation function. A key difference, however, with backpropagation is that intermediate calculations do not need to be kept in memory, because the learning is local per layer. This means that memory allocated for matrices can be re-used, which is facilitated by the linear algebra library by being able to adjust matrix dimensions. This makes networks trained with forward-forward easier to scale up in depth.
After a forward pass, the loss is calculated for the layer and the gradient is calculated with respect to the layer weights and biases. Then, the weights and biases for that layer are updated using SGD.

4.1.3. Transfer Learning

Network Topologies & Hyperparameters

The network topologies that are used for transfer learning are different than the other topologies discussed. While it would be possible for example to pre-train only one of the layers of the dual hidden layer network that was discussed earlier, this would not be a realistic use-case for transfer learning. Instead, a simple convolutional feature extractor is pre-trained (see fig. 4.3), on top of which a classification network can be trained. The convolutions are 3x3 kernels and no padding is applied. In between convolutional layers, max-pooling is applied. The classification networks are a single hidden layer network for backpropagation and a single layer network for forward-forward. The hyperparameters for these classification networks stay the same. For forward-forward, the feature vector is extended with the one-hot encoded labels. The use of convolutional neural networks means that no direct comparison can be made with the other networks. Instead, the aim is to show the impact on performance that adding such a network can have.

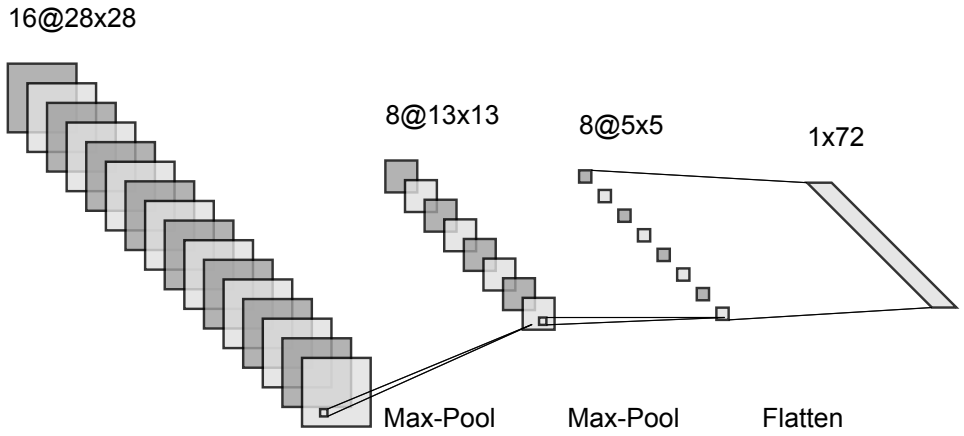


Figure 4.3: Feature extraction network for transfer learning. The network reduces the number of features from 784 to 72 using a convolutional network. The kernel size is 3x3 and no padding is applied (hence 28x28 becomes 26x26 after convolution, etc.). A classification network can be added on top of this feature extractor, which can be trained with backpropagation or forward-forward.

4.2. Testing Procedures & Metrics

For all learning algorithms, the fashion MNIST dataset is used for training. The following metrics are used to measure performance.

- Test accuracy after 1 epoch
- Test accuracy after 5 epochs
- Test accuracy after 10 epochs
- Required memory (kB)
- Inference time per example (μs)
- Training time per epoch (s)
- Energy per inference (mJ)

The metrics related to accuracy (after 1, 5, 10 epochs) are chosen to represent different stages of the optimization procedure. The choice to have no accuracy measurements beyond 10 epochs is made because, although some training still takes place after 10 epochs, it is to a much lesser degree than between epochs 1-10 (for example, see fig. B.14). Running all algorithms for many epochs also takes a considerable amount of time on the MCU (see tables 5.1 and 5.3).
Metrics regarding the accuracy and inference/training times are automatically collected using a Python script. The script automatically trains the network that is loaded onto the microcontroller for 10 epochs and measures accuracy after each epoch, allowing for accuracy over time plots. This is, however, not possible for multi-layer networks trained using forward-forward, because the layers are sequentially trained for a certain amount of epochs. Instead, the accuracy is measured at 1, 5 and 10 epochs.
For multi-layer networks trained using forward-forward, the training time per epoch is not directly defined, because all layers are trained for a certain amount of epochs sequentially. The training time per epoch is instead defined as the sum of training times per epoch of all the individual layers. This makes the training time per epoch scale similarly to other training procedures.
All time-related metrics exclude data transfer times. This is done to exclude variance coming from SD card reading, thus making the results more generalizable to other microcontrollers.

In addition to these metrics, in order to compare the adaptability of the models, a dataset with a moving distribution is created from the fashion MNIST dataset. Of one of the classes, 90% of the training data is initially removed from the dataset. The models (trained with offline backpropagation, online backpropagation, and online forward-forward) are then tested to establish a baseline accuracy. The online trained models are then trained on the full dataset, after which the accuracy is again measured. The idea is that the accuracy should increase, which indicates that the models are able to adapt to new environments.

The mean of the power consumption is measured once, as it is observed to be the same across algorithms. Using the power and the inference time, the energy consumption per inference is calculated.

5

Results and Discussion

In this section, the results of backpropagation, forward-forward, and transfer learning are presented. For backpropagation and forward-forward, detailed results of the tested network topologies and training algorithm are given, followed by a discussion and a comparison of relevant results obtained by the software and FPGA teams. More information and full-size versions of the figures are available in appendix B. All reported accuracies are on the Fashion MNIST test dataset and all reported times exclude data transfer.

5.1. Performance of backpropagation

In this section, we discuss the performance of various configurations of the backpropagation algorithm using the Fashion MNIST dataset. We begin by establishing a baseline, then investigate the impact of learning rate decay and momentum. Finally, we test the performance of a dual-layer network and compare all these configurations. Inference-only TensorFlow lite models are also tested for comparison.

Table 5.1: Collected metrics for on-device backpropagation. Memory usage between parentheses is using the memory-optimized version of SGD as described in chapter 4.

Metrics	**Baseline**	**Baseline + LR Decay**	**Baseline + Momentum**	**Dual layer**	**Dual layer + LR Decay**
Accuracy on test set after 1 epoch	0.7999	0.8404	0.8154	0.8006	0.8502
Accuracy on test set after 5 epochs	0.8454	0.8591	0.8549	0.8569	0.8649
Accuracy on test set after 10 epochs	0.8563	0.8648	0.8670	0.8684	0.8705
Memory required (kB) Variables	265.22 (165.81)	265.22 (165.81)	265.22	278.47 (173.94)	278.47 (173.94)
Memory required (kB) Code	99.90	99.90	100.52	102.02	102.02
Avg. inference time (us) / image	258.4	258.4	258.3	270.0	270.0
Training time (s) / epoch	37.29	37.29	37.86	39.66	39.66
Energy (mJ) / inference	0.171	0.171	0.171	0.179	0.179

5.1.1. Baseline

Our baseline network for testing the performance of the backpropagation algorithm has a topology of 784-32-10, reflecting the input layer of 784 nodes (equivalent to the dimensionality of an image in the Fashion MNIST dataset), a hidden layer of 32 nodes, and an output layer of 10 nodes (corresponding to the 10 classes of the dataset). The learning rate for this configuration is a constant 0.01, and we do not use momentum.

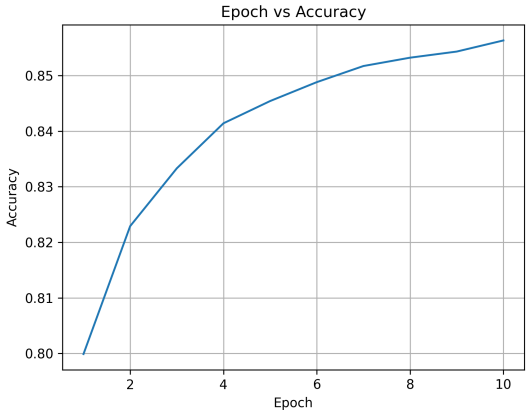


Figure 5.1: Accuracy of Baseline over 10 epochs

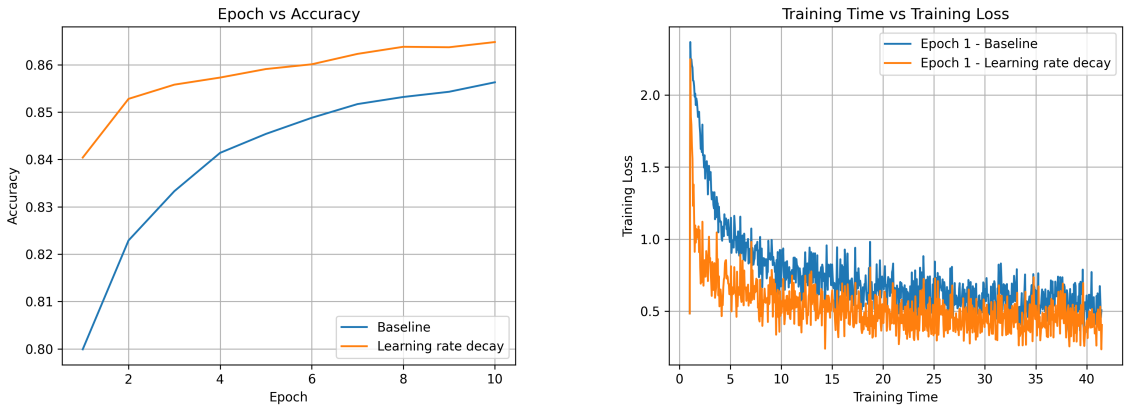
As can be seen from the results, the accuracy on the test set of the model steadily increases with each epoch, indicating that the model is learning effectively from the training process.

In terms of memory utilization, the baseline configuration required a total of 365.11 kibibytes (KiB). This was divided between memory required for variable storage (265.22 KiB) and code (99.90 KiB).

The baseline configuration will provide us with a reference point against which the performance and resource utilization of the other configurations can be compared. In the following subsections, we discuss the impact of introducing learning rate decay, momentum, and a dual-layer network architecture.

5.1.2. Learning rate decay

To investigate the impact of learning rate decay, we use the same network topology (784-32-10), but this time we implement a learning rate decay strategy. The initial learning rate is set to 0.1. After every 200 batches, the learning rate is multiplied by 0.95, with the condition that it cannot decrease below 0.01. Momentum is not employed in this configuration. We compare the performance of this network with the baseline to understand how learning rate decay affects the training process.



(a) Accuracy over 10 epochs

(b) Loss during the first epoch (Loss vs Time (s))

Figure 5.2: Baseline vs Learning rate decay configuration

First, looking at the accuracy, we see that the model with learning rate decay starts with a higher accuracy after the first epoch compared to the baseline (0.8404 vs 0.7999). This trend continues throughout the training period with the learning rate decay configuration consistently achieving higher accuracies at the end of each epoch. After 10 epochs, the model with learning rate decay reached an accuracy of 0.8648, while the baseline model reached an accuracy of 0.8563. This indicates that learning rate

decay can contribute to achieving better accuracies in fewer epochs. However, with a lot of training cycles, both methods will find similar local minima, thus making their accuracies converge.

The training times for the baseline and the baseline + learning rate decay are the same, demonstrating that the inclusion of learning rate decay does not have an impact on computation time.

5.1.3. Momentum

Next, we introduce momentum into our configuration. The network topology remains the same (784-32-10). The momentum is set to 0.9, without learning rate decay. Training with momentum is compared with training with learning rate decay in fig. 5.3.

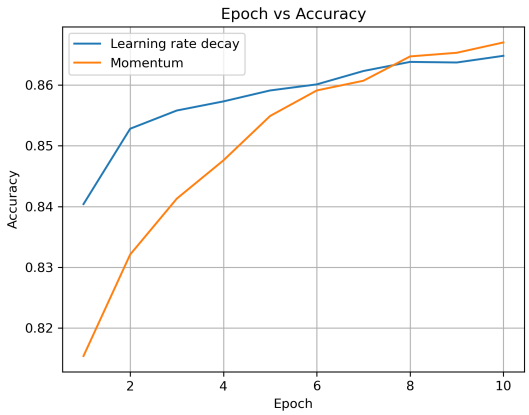


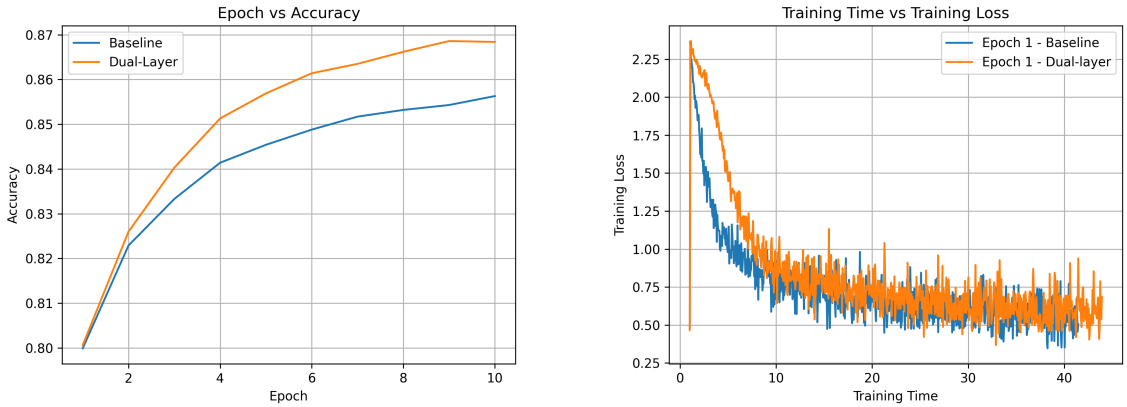
Figure 5.3: Accuracy of training with Learning rate decay vs Momentum configuration over 10 epochs

In the plot, it can be seen that the configuration with learning rate decay starts out with a higher accuracy. This is due to the higher learning rate in the beginning, compared to the configuration with momentum. As the training proceeds, it can be seen that the configuration with momentum learns quicker than the configuration with learning rate decay and learning rate decay gets overtaken by momentum by epoch 8. Both configurations finish training with similar accuracies (0.8670 for momentum and 0.8648 for lr decay), with a slight edge to the momentum configuration.

Both learning rate decay and momentum improve the accuracy of the model compared to the baseline without increasing test time.

5.1.4. Dual-Layer network

In this configuration, we experiment with a more complex network architecture: a dual-layer network with a topology of 784-32-32-10. This represents an input layer of 784 nodes, two hidden layers of 32 nodes each, and an output layer of 10 nodes. We revert to a constant learning rate of 0.01 and do not employ momentum or learning rate decay. By comparing the performance of this dual-layer network with the simpler single-hidden-layer networks, we can gain insights into the trade-offs involved in increasing the complexity of the network.



(a) Accuracy over 10 epochs

(b) Loss during the first epoch

Figure 5.4: Baseline vs Dual-Layer configuration

Comparing the final accuracies achieved at the end of the 10th epoch, the baseline network reached an accuracy of 0.8563, while the dual-layer network achieved a higher accuracy of 0.8684. Because the dual-layer network has a greater number of parameters and hence a higher capacity to model complex patterns, it seems to provide a performance advantage over the simpler network.

A notable aspect here is the increase in inference time for the dual-layer network. The inference time per example for the dual-layer network (270.0 µs) is consistently higher than the baseline network (258.4 µs). This indicates that increasing the complexity of the network in this manner incurs a computational cost.

To keep a fair comparison, and only vary one parameter at a time, only the topology was varied compared to the baseline. However, this is not optimal because the dual-layer network requires a larger initial learning rate to tune the increased parameter dependencies, as can be seen from fig. 5.4b.

Nevertheless, to demonstrate what the dual-layer network can achieve, the same network is trained using learning rate decay, with an initial learning rate of 0.1 and a decay of 0.95 every 200 batches to a minimum of 0.01.

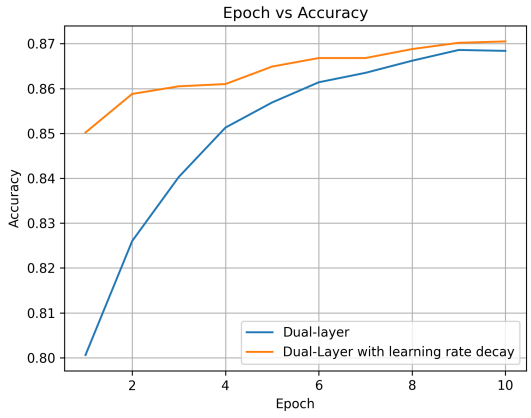


Figure 5.5: Accuracy of Dual-layer without learning rate decay vs Dual-layer with learning rate decay configuration over 10 epochs

The implementation of a learning rate decay schedule has significantly enhanced the dual-layer network's performance during the initial training phase, as can be seen from fig. 5.5. From the results, the dual-layer network with learning rate decay achieved a final accuracy of 0.8705 after 10 epochs,

slightly better than the dual-layer network without learning rate decay, which reached an accuracy of only 0.8684. The learning rate decay allows the network to converge significantly faster.

5.1.5. Larger networks using memory-optimized SGD

In this section, we present the results from an optimized version of the Stochastic Gradient Descent (SGD) algorithm that allows for larger network sizes, such as a 784-80-80-10 topology. This larger network is trained using learning rate decay but without momentum.

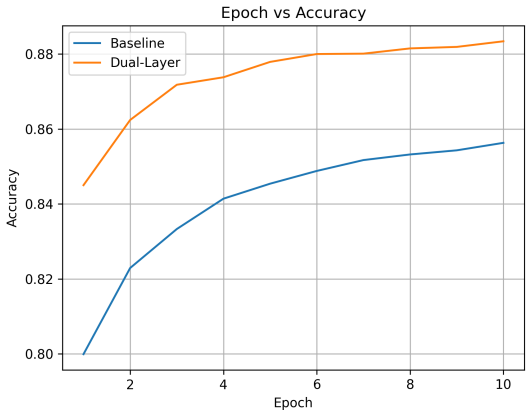


Figure 5.6: Accuracy of Baseline vs Maximum Dual-Layer configuration over 10 epochs

As seen from the results, the maximal network achieved a final accuracy of 0.8834 after 10 epochs. This is the highest score tested, and considerably higher than the final accuracy of 0.8563 that the baseline model achieved within the same number of epochs. This suggests that the optimized SGD algorithm successfully leveraged the additional capacity provided by the larger network size to learn more complex patterns in the data, resulting in improved predictive performance.

When comparing the memory utilization of the maximum network with the baseline model, there is a noticeable increase in the memory footprint. The maximum network uses approximately 356.19 KiB of memory for variables, which is an increase compared to the 265.22 KiB utilized by the baseline model. This increase in memory use is primarily due to the larger size of the maximal network, which requires more memory to store the additional weights and biases of the extra neurons. However, it shows an impressive decrease in memory required by each neuron compared to the non-optimized version.

The memory used for the code is almost similar for both networks, with the maximum network using a slightly larger amount (102.21 KiB) compared to the baseline model (99.90 KiB).

5.1.6. Comparison with TF Lite

Table 5.2: Collected metrics for inference-only TensorFlow Lite models (same hyperparameters as baseline models). Single layer model is 784-32-10, dual layer model is 784-32-32-10.

Metrics	Single layer	Dual layer
Accuracy after 10 epochs	0.8560	0.8631
Memory required (kB) Variables	52.12	54.13
Memory required (kB) Code	105.18	105.18
Avg. inference time (us) / image	76.9	82.1
Energy (mJ) / inference	0.051	0.054

From the tables table 5.1 and table 5.2 it can be seen that the on-device training algorithms can achieve similar behavior as the Tensorflow quantized models.

- **Accuracy**: It is observable from the tables that TensorFlow Lite models achieved a slightly lower accuracy compared to the on-device trained models, with the dual layer model performing the best in both scenarios.
- **Memory Usage**: The memory required for variables was significantly lower in TensorFlow Lite models. For instance, the single-layer model required only 52.12 kB as compared to 265.22 kB (or 165.81 kB for the memory-optimized version) for the on-device trained model.
- **Inference Time**: TensorFlow Lite models exhibited a faster inference time per image, taking around 76.9 - 82.1 microseconds, compared to 258.4 - 270.0 microseconds for the on-device models.
- **Energy Usage**: The energy consumed per inference was lower for the TensorFlow Lite models.

The comparison reveals that on-device training may seem unnecessary when looking at accuracy and memory utilization. Nonetheless, on-edge devices often gain considerable advantages from their capacity to adapt to changing circumstances.

On device learning
Consider a scenario where we alter our dataset by removing 90% of all the images from the t-shirt class in the Fashion MNIST dataset. We employ the baseline 784-32-10 network for this experiment. Upon training this network for five epochs, it yields an accuracy of 85.53% on the reduced dataset, slightly better than the baseline's 84.54%. However, when the network trained on the reduced dataset is tested exclusively on t-shirt images, it only achieves an accuracy of 35.22%. In contrast, the baseline retains its original accuracy of 84.54%. This illustrates that the network has not effectively learned the distinctive features of t-shirts and thus fails to categorize them correctly.

It's noteworthy that all the prior networks, whether trained on the edge or as quantized Tensorflow models, could have achieved comparable accuracy levels. However, the unique advantage of on-edge learning becomes apparent when we allow our network (previously trained on the reduced dataset) to undergo additional training on the original, unaltered dataset. After just five more epochs, it manages to classify t-shirts with an accuracy of 81.95%, marking an improvement of more than 46%.

This is a great illustration of the necessity for on-edge learning, albeit an artificial one. For potential applications see chapter 1, and we leave it to the reader to imagine many more.

5.1.7. Discussion

The previous sections have extensively explored several strategies to optimize the backpropagation algorithm in different neural network architectures. The introduction of learning rate decay showed clear benefits. For instance, in the case of the single-layer network with 32 nodes, accuracy after 10 epochs increased from 0.8563 to 0.8648 when learning rate decay was introduced. Similar gains in accuracy were observed for the dual-layer network with learning rate decay, where accuracy improved from 0.8006 to 0.8502 after the first epoch.

Introducing momentum in the single-layer network with 32 nodes yielded slightly more accuracy gains, increasing from 0.8563 to 0.8670 after 10 epochs. This suggests that momentum offered a better optimization. It does, however, come at the cost of not being able to make the SGD memory optimization suggested in chapter 4.

Regarding the network architectures, it's worth noting that although the dual-layer network with learning rate decay achieved the highest accuracy after 10 epochs (0.8705), this configuration also required the highest memory utilization (278.47 kB for variables and 102.02 kB for code) and had a longer inference time per image (270 microseconds) compared to the other configurations. This indicates a trade-off between model complexity, computational requirements, and prediction accuracy.

Training time per epoch stayed the same with learning rate decay. It increased slightly with the introduction of momentum and significantly with an additional hidden layer, as one might expect due to the increased complexity of computations.

5.2. Performance of Forward-Forward Algorithm

In this section, we discuss the performance of various configurations of the Forward-Forward algorithm using the Fashion MNIST dataset. We begin by establishing a baseline, then investigate the impact of learning rate decay. Finally, we test the performance of a dual-layer network and compare all these configurations. A direct comparison between BP and FF can be found in table B.19.

Table 5.3: Forward-Forward Model Comparison. Memory usage between parentheses is using the memory-optimized version of SGD as described in chapter 4. Training time for multi-layer configurations is defined as the sum of training times per layer.

Metrics	Baseline	FF Dual-layer
Accuracy on test set after 1 epoch	0.6932	0.7002
Accuracy on test set after 5 epochs	0.8226	0.8193
Accuracy on test set after 10 epochs	0.8374	0.8206
Memory required (kB) Variable	267.06 (168.94)	274.19 (176.06)
Memory required (kB) Code	98.09	99.21
Avg. inference time (us) / image	2533.3	2794.3
Training time (s) / epoch	71.18	106.30
Energy (mJ) / inference	1.676	1.848

5.2.1. Baseline

The forward-forward algorithm was tested with a single-layer network, having a topology of 784-32. This configuration does not use learning rate decay or momentum.

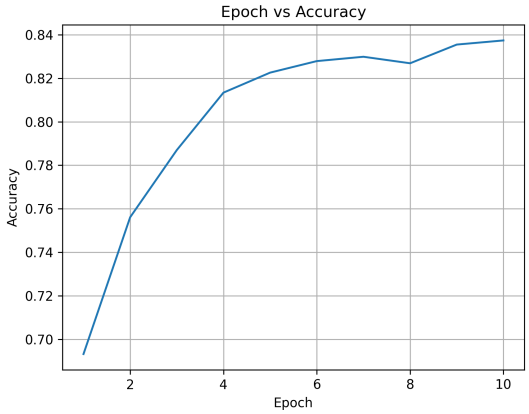


Figure 5.7: Accuracy of Baseline over 10 epochs

The forward-forward algorithm shows steady improvements in accuracy over the 10 epochs (as can be seen in fig. 5.7). After the first epoch, the accuracy was measured at 0.6932. By the end of the fifth epoch, the model accuracy improved significantly to 0.8226. Eventually, after 10 epochs, the accuracy was noted at 0.8374. This indicates that the forward-forward algorithm can provide substantial learning capacity to the neural network.

However, one should note that the test time for this algorithm was quite high, approximately 26.95 seconds for each epoch. This can be attributed to the fact that the network has to run a forward pass on every label, and select the label with the highest output. The substantial increase in test time suggests that while the forward-forward algorithm can improve accuracy over time, it does so at the cost of computational speed.

In terms of memory utilization, the forward-forward algorithm with a single-layer network required 267.06 kB for variables, which is slightly more than the backpropagation baseline in the previous sections. The code size measured slightly lower at 98.09 kB. This is to be expected, as the memory savings that forward-forward has when compared to backpropagation only show up at higher network depths.

5.2.2. Dual-layer network

In the dual-layer configuration, each layer is comprised of 32 neurons. Surprisingly, after multiple epochs, the performance of the dual-layer model is worse than that of the single-layer model. When further probing the second layer (by only taking into account its activations for accuracy measurements), however, it is found that the second layer does learn (its accuracy is 0.7327). Furthermore, in table B.18, a batch size of 32 is used. Here, the dual-layer model outperforms the single-layer model.

In terms of memory utilization, the dual-layer network required slightly more memory for storing variables, measuring 274.19 kB compared to 267.06 kB for the single-layer configuration. This is a difference of 7.13 kB, compared to the 13.05 kB for backpropagation. This difference exists because there is no need to store the gradient and neuron activations across multiple layers. This is a significant advantage of the Forward-Forward algorithm. The dual-layer configuration for forward-forward uses less memory than the dual-layer configuration for backpropagation (274.19 vs 278.47). This trend will continue as the network depth is increased.

5.2.3. Comparison with TF Lite

Table 5.4: Collected metrics for inference-only TensorFlow Lite models trained and quantized using forward-forward by the software team. Single layer model is 784-32, dual layer model is 784-32-32

Metrics	Single layer	Dual layer
Maximal achieved accuracy	0.6966	0.6793
Memory required (kB) Variables	52.09	60.47
Memory required (kB) Code	105.18	105.18
Avg. inference time (us) / image	1007.7	1188.7
Energy (mJ) / inference	0.665	0.786

From Tables 5.3 and 5.4, it can be observed that the on-device trained model and the Tensorflow Lite model show substantially different behaviour.

- **Accuracy**: The on-device trained models achieved higher accuracy than the TensorFlow Lite models, with the single-layer on-device model performing the best after 10 epochs.
- **Memory Usage**: Similar to the Backpropagation comparison, TensorFlow Lite models require significantly less memory for variables.
- **Inference Time**: TensorFlow Lite models still have a shorter inference time per image compared to the on-device trained models.
- **Energy Usage**: Energy consumption per inference for TensorFlow Lite models is lower compared to on-device trained models, because of their reduced inference time.

5.2.4. Discussion

The dual-layer network implemented with the forward-forward algorithm performs worse than the single-layer network. This is unexpected, but in line with the results from the software team. In table B.18 it is shown that it is possible for the dual-layer model to outperform the single-layer model. This seems to indicate that the dual-layer model can only outperform the single-layer model using specific hyperparameters.

The TF Lite models for forward-forward performed significantly worse than the models trained on-device, however the reported accuracy by the software team for the dual layer model on full precision is 83%. This is more or less in line with the result for on-device forward-forward (82%). Therefore, it can be assumed that the quantization process drastically affected performance.

When compared to the offline trained models using backpropagation, the models trained with forward-forward fall short on all metrics:

- **Accuracy:** Like before, it is hypothesized that the low accuracies are due to the quantization process.

- **Memory:** The memory advantage of forward-forward only exists during training, as for inference, no intermediate steps need to be remembered, because backpropagation is not done for inference. Furthermore, the memory usage for the network trained with forward-forward is higher than for backpropagation, because in order to obtain individual layer activations, it requires a TF Lite Micro interpreter for each layer.
- **Inference time:** The same disadvantages for inferring models trained with forward-forward still exist: for every class, the network needs to be inferred once per example. This results in much higher inference times when compared to the models trained with backpropagation.
- **Energy per inference:** Because the inference time is higher and the power is constant under load, the energy per inference scales linearly with inference time and is thus very high.

This makes inference of models trained using forward-forward with TF Lite hard to justify, even if the accuracy were comparable to other models.

5.3. Transfer Learning

Transfer learning is implemented with the feature extractor network and classification heads, as described in chapter 4. For backpropagation, the classification head is a 72-32-10 network and for forward-forward it is a 82-32 network. The results are shown in table 5.5.

Table 5.5: Forward-Forward Model Comparison. Memory usage between parentheses is using the memory-optimized version of SGD as described in chapter 4.

Metrics	Backpropagation	Forward-Forward
Accuracy on test set after 1 epoch	0.8552	0.7491
Accuracy on test set after 5 epochs	0.8672	0.8298
Accuracy on test set after 10 epochs	0.8717	0.8511
Memory required (kB) Variable 1	78.31 (67.90)	84.31 (73.93)
Memory required (kB) Code	154.98	156.91
Avg. inference time (us) / image	2425.5	2681.5
Training time (s) / epoch	147.91	151.82
Energy (mJ) / inference	1.604	1.774

As can be seen from the results, both learning algorithms were able to achieve higher accuracies than their baseline implementation. Backpropagation was able to learn more effectively from the features output by the convolutional neural network, while also using less memory. Training was slower than the baseline in both training and inference because the feature extractor needed to be inferred for each example. The memory taken up by the code is significantly higher than for backpropagation or forward-forward alone, due to the extra code required for the TensorFlow Lite Micro interpreter and feature extractor model. The memory taken up by variables, however, is significantly lower, because the weight matrices of the classification heads are reduced in size since the amount of input features is reduced from 784 to 72.

5.4. Comparison with other teams

In table 5.6, a comparison between backpropagation baseline models of the software, microcontroller, and FPGA teams is shown. The accuracies for the full-precision TensorFlow model and the on-device trained model are similar, which is expected. Furthermore, the accuracies of the quantized models are similar for all teams. Training times per epoch are around 7.9 times higher on the MCU than on the PC. Inference times are only 4 times higher. Interestingly, the inference time reported by the FPGA team (16.32 µs) is much lower than the other int8 quantized models (34.62 µs using TF Lite on the PC, 70.19 µs using TF Lite on the microcontroller).
A comparison between the software team and microcontroller team is shown in table 5.7. Both models achieve similar accuracies. Training on the MCU takes 2.4 times as long on the MCU as it does on the PC. Inference takes 4.2 times as long. All obtained results can be found in B.3. The Dual Layer FF model is the only model whose test accuracy differs more than 2% (2.13%) between the TF and on-device implementation.

Table 5.6: Comparison table of the BP baseline model (784-32-10, no LR decay, no momentum, batch size = 16, lr = 0.01) between the different groups. It can be seen that the model trained with on-device BP and SGD performs similarly to the model trained using TensorFlow. The quantized model also performs similarly to the models running on the PC and FPGA.

Metrics	**TF (PC)**	**TF Lite (PC)**	**TF Lite (MCU)**	**Local MCU**	**FPGA**
Datatype	fp32	int8	int8	fp32	int8
Accuracy on test set after 1 epochs	80.72	NA	NA	79.99	NA
Accuracy on test set after 5 epochs	85	NA	NA	84.54	NA
Accuracy on test set after 10 epochs	85.82	NA	85.6	85.63	NA
Accuracy on test set after 30 epochs	86.41	86.33	86.33	NA	85.53
Memory required (KiB) (Variable | Code)	NA	NA	60.16 | 105.74	265.22 | 99.90	25.08 | 1.34
Inference time per sample(us)	65.02	34.62	70.19	258.4	16.32
Training time (s)	142.2	NA	NA	372.9	NA
Training time per epoch (s)	4.74	NA	NA	37.29	NA

Table 5.7: Comparison table of the FF baseline model (784-32, no lr decay, no momentum, batch size = 16, lr = 0.1) between the software and MCU groups. Accuracies are similar between models trained on the MCU and models trained using TensorFlow.

Metrics	**TF (PC)**	**Local MCU**
Datatype	fp32	fp32
Accuracy on test set after 1 epochs	68.72	69.32
Accuracy on test set after 5 epochs	79.50	82.26
Accuracy on test set after 10 epochs	82.55	83.74
Memory required (KiB) (Variable | Code)	NA	318.19 | 98.09
Inference time per sample(us)	604.1	2533.27
Training time per epoch (s)	29.24	71.18

5.5. Discussion

The Backpropagation and Forward-Forward algorithms can both be used in the field of neural networks on microcontrollers, but they function quite differently and therefore have distinct strengths and limitations. Here, we offer a discussion comparing these two algorithms based on their implementation in the context of this research.

1. Accuracy: In terms of accuracy on the test set, backpropagation consistently achieved higher accuracy across the board, both for single-layer and dual-layer configurations. For instance, after 10 epochs, a single-layer backpropagation network achieved 0.8563 accuracy, while a single-layer FF network reached only 0.8374. When adding a second layer to the network, the FF algorithm's accuracy was 0.8206, falling short of the backpropagation's single-layer result.

2. Memory utilization: Comparing the memory utilization of the two algorithms, backpropagation requires more memory for storing variables at higher network depths (with two layers, BP uses 1.56% more memory than FF, but this difference grows bigger as more layers are added). This can be attributed to the fact that backpropagation needs to keep track of intermediate values in the computational graph for use during the backward pass, whereas FF only needs to maintain values for the forward pass. Thus, in environments where memory is a crucial constraint, the FF algorithm could be a better choice.

3. Training and Inference Time: Concerning computational time, backpropagation proved to be more efficient. In terms of training time per epoch, backpropagation, even with the dual-layer network, was faster compared to FF. The inference time of backpropagation was also significantly lower. This could be an essential factor to consider in some real-world applications where time efficiency is vital.

6

Conclusion

6.1. Summary of Findings

While backpropagation shows higher accuracy and efficiency, it comes with higher memory requirements and complexity. On the other hand, the forward-forward algorithm, despite its lower accuracy, can be more memory-efficient and simpler to implement. The choice between these two algorithms largely depends on the specific requirements and constraints of the problem at hand. It's also worth noting that these results might be improved by careful hyperparameter tuning, which could potentially enhance both algorithms' performance.

The project has fulfilled all functional requirements outlined in chapter 2. Namely, all required algorithms and models have been implemented. Furthermore, the project has fulfilled all mandatory requirements:

1. The software for the microcontroller must be written in C or C++.
    - The software has first been implemented in C, targeting x86 systems. Later, it was ported to C++, targeting the ARM-based Teensy.
2. All online training algorithms must use less than 512 kB RAM.
    - The developed algorithms use no more than 512 kB RAM and can therefore run on the Teensy and many other microcontrollers.
3. Test accuracy after full training on Fashion MNIST should be over 80%.
    - All networks were able to achieve an accuracy of over 80%, both using backpropagation and forward-forward, except for the TensorFlow Lite models using forward-forward provided by the Software group. It is suspected that accuracy is lost during the quantization process, since the full precision model does reach the target.
4. Accuracy on-device learning should be nearly identical to full precision TensorFlow models (within 2%).
    - The test accuracy of the on-device trained networks is within 2% of the test accuracy achieved by the full-precision TensorFlow models, except for the dual layer forward-forward configuration. The accuracy is, however, very close to this target (2.13%). Furthermore, most accuracies of BP configurations are within 1% of each other.
5. Implement Backpropagation and Forward-Forward on the MCU
    - Backpropagation and Forward-Forward have been successfully implemented and tested on the Teensy microcontroller board.

Some, but not all of the trade-off requirements have been fulfilled:

1. Minimize the memory use of all algorithms, striving for a target lower limit of 256 kB RAM - a common memory size for microcontrollers.
    - Memory use has been reduced using multiple optimizations. For example, an optimization to SGD allowed for in-place weight updating, without the need for gradient matrices. Only backpropagation baseline, baseline + LR Decay, TensorFlow Lite models, and transfer learning models can fit in less than 256 kB RAM.
2. Minimize the time required for training, aiming to achieve a goal of less than 1 minute per epoch.
    - Some effort has been put into tuning the hyperparameters for convergence speed, such as for forward-forward, but the training time is mostly unoptimized. Still, all baseline and dual-layer backpropagation models were able to train one epoch in less than 1 minute.
3. Minimize inference time, targeting a maximum duration of 0.5 seconds per batch of 16 images.
    - Minimizing inference time has not been a major focus of this study, but all models can be inferred on a batch of 16 images in less than 0.5 seconds.
4. Maximize the accuracy of trained networks on the MNIST test dataset. This requirement does not have a specified target but should always be optimized.
    - Effort was put into maximizing the accuracy of the trained networks, in particular the networks trained with backpropagation. Experiments included LR decay and momentum. These improved the accuracy of the models.
5. Experimenting with different topologies, with a minimum of online single-layer.
    - Only single-layer and dual-layer networks were explored in this study.

6.2. Limitations of the Study

This study has focused on studying the viability of training neural networks with the forward-forward algorithm and comparing this to backpropagation. We identify several limitations that may have influenced the outcomes and interpretations of our findings.

1. The study was conducted using a specific microcontroller board, the Teensy 4.1, and a specific dataset, Fashion MNIST. The results, therefore, may not be generalizable to other microcontrollers and datasets.
2. Extensive hyperparameter tuning was not attempted, which can have a significant impact on the performance (in terms of our trade-of requirements) of the models. While we can draw the conclusion that both algorithms are viable, nothing can be said about the maximal achievable accuracy.
3. The Forward-Forward algorithm is a relatively new area of research, with limited prior work done in the context of low-cost hardware. The implementation and testing of the Forward-Forward algorithm were based on our understanding of the algorithm at the time of the study, and there may be room for improvements or different approaches to its implementation. These could significantly improve memory utilization and accuracy.
4. Memory usage has been optimized such that it is at least no more than 512 kB, but the implementations may not be optimal in terms of memory use.
5. This study has only focused on training with full precision. No conclusions can be drawn on how the discussed algorithms may behave when quantized training is applied.

6.3. Recommendations for Future Work

Recommendations for future studies include:

1. Identifying multi-layer training issues with forward-forward
    - In this thesis, it was shown that forward-forward has reduced memory usage when compared to backpropagation if the depth of the network is increased. However, it was also shown that the multi-layer forward-forward setup performed worse than the single-layer setup. This means that in order to exploit the strength of forward-forward this issue will have to be addressed.
2. Quantized forward-forward training on microcontrollers
    - Studies have already been done on quantized backpropagation for on-device training, but the same has not been done for forward-forward. Quantized training using forward-forward could allow for increased network sizes.
3. Evaluation on Real-World Datasets
    - While the Fashion MNIST dataset was used in this study due to its standardization and ubiquity in similar works, a more realistic assessment of on-edge learning would involve using diverse, real-world datasets.

Bibliography

[1] X. Wang, X. Ren, C. Qiu, Z. Xiong, H. Yao, and V. C. M. Leung, "Integrating edge intelligence and blockchain: What, why, and how," *IEEE Communications Surveys Tutorials*, vol. 24, no. 4, pp. 2193–2229, 2022. DOI: 10.1109/COMST.2022.3189962.

[2] J. Hartmann, P. Cappelletti, N. Chawla, F. Arnaud, and A. Cathelin, "Artificial intelligence: Why moving it to the edge?" In *ESSDERC 2021 - IEEE 51st European Solid-State Device Research Conference (ESSDERC)*, 2021, pp. 1–6. DOI: 10.1109/ESSDERC53440.2021.9631778.

[3] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, 2019, ISSN: 00189219. DOI: 10.1109/JPROC.2019.2922285.

[4] J. Dalzochio, R. Kunst, E. Pignaton, *et al.*, "Machine learning and reasoning for predictive maintenance in industry 4.0: Current status and challenges," *Computers in Industry*, vol. 123, p. 103 298, 2020, ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2020.103298. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361520305327.

[5] N. Davari, B. Veloso, R. P. Ribeiro, P. M. Pereira, and J. Gama, "Predictive maintenance based on anomaly detection using deep learning for air production unit in the railway industry," IEEE, Oct. 2021, pp. 1–10, ISBN: 978-1-6654-2099-0. DOI: 10.1109/DSAA53316.2021.9564181. [Online]. Available: https://ieeexplore.ieee.org/document/9564181/.

[6] O. Debauche, S. Mahmoudi, M. Elmoulat, S. A. Mahmoudi, P. Manneback, and F. Lebeau, "Edge ai-iot pivot irrigation, plant diseases, and pests identification," *Procedia Computer Science*, vol. 177, pp. 40–48, 2020, The 11th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2020) / The 10th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2020) / Affiliated Workshops, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.10.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920322742.

[7] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," *IEEE Sensors Journal*, vol. 22, pp. 21 362–21 390, 22 2022, ISSN: 1558-1748. DOI: 10.1109/JSEN.2022.3210773.

[8] F. Sakr, F. Bellotti, R. Berta, and A. D. Gloria, "Machine learning on mainstream microcontrollers," *Sensors 2020, Vol. 20, Page 2638*, vol. 20, p. 2638, 9 May 2020, ISSN: 1424-8220. DOI: 10.3390/S20092638. [Online]. Available: https://www.mdpi.com/1424-8220/20/9/2638/htm%20https://www.mdpi.com/1424-8220/20/9/2638.

[9] "Teensy® 4.1." (), [Online]. Available: https://www.pjrc.com/store/teensy41.html.

[10] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," Aug. 2017. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist.

[11] J. Zupan, "Introduction to artificial neural network (ann) methods: What they are and how to use them," *Acta Chimica Slovenica*, vol. 41, pp. 327–327, 1994.

[12] L. Noriega, "Multilayer perceptron tutorial," *School of Computing. Staffordshire University*, vol. 4, p. 5, 2005.

[13] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin, "Backpropagation: The basic theory," *Backpropagation: Theory, architectures and applications*, pp. 1–34, 1995.

[14] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[15] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*, Ieee, 2017, pp. 1–6.

[16] R. Rojas and R. Rojas, “The backpropagation algorithm,” *Neural networks: a systematic introduction*, pp. 149–182, 1996.

[17] L. Bottou, “Stochastic gradient descent tricks,” *Neural Networks: Tricks of the Trade: Second Edition*, pp. 421–436, 2012.

[18] N. Ketkar and N. Ketkar, “Stochastic gradient descent,” *Deep learning with Python: A hands-on introduction*, pp. 113–132, 2017.

[19] E. Zamora and H. Sossa, “Dendrite morphological neurons trained by stochastic gradient descent,” *Neurocomputing*, vol. 260, pp. 420–431, 2017.

[20] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, “On-device training under 256kb memory,” Jun. 2022. [Online]. Available: https://arxiv.org/abs/2206.15472.

[21] F. Ortega-Zamorano, J. M. Jerez, D. U. Munoz, R. M. Luque-Baena, and L. Franco, “Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, pp. 1840–1850, 9 Sep. 2016, ISSN: 2162-237X. DOI: 10.1109/TNNLS.2015.2460991.

[22] G. Hinton, *The forward-forward algorithm: Some preliminary investigations*, 2022. arXiv: 2212.13345 [cs.LG].

[23] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, “Pruning and quantization for deep neural network acceleration: A survey,” 2021.

[24] H. Ren and T. A. Runkler, “Tinyol: Tinyml with online-learning on microcontrollers; tinyol: Tinyml with online-learning on microcontrollers,” 2021. DOI: 10.1109/IJCNN52387.2021.9533927.

[25] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[26] I. Deligiannis and G. Kornaros, “Adaptive memory management scheme for mmu-less embedded systems,” IEEE, May 2016, pp. 1–8, ISBN: 978-1-5090-2282-3. DOI: 10.1109/SIES.2016.7509439.

[27] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, “Scalable methods for 8-bit training of neural networks,” May 2018.

[28] “Makemore.” (), [Online]. Available: https://github.com/karpathy/nn-zero-to-hero/blob/master/lectures/makemore/makemore_part4_backprop.ipynb (visited on 06/13/2023).

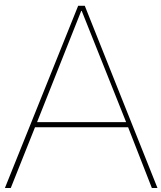
[29] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, *Integer quantization for deep learning inference: Principles and empirical evaluation*, 2020. arXiv: 2004.09602 [cs.LG].

[30] “Pytorch forward forward.” (), [Online]. Available: https://github.com/mohammadpz/pytorch_forward_forward (visited on 06/08/2023).

[31] “Github - zalandoresearch/fashion-mnist: A mnist-like fashion product database. benchmark.” (), [Online]. Available: https://github.com/zalandoresearch/fashion-mnist.

[32] “Eigen linear algebra library.” (), [Online]. Available: https://eigen.tuxfamily.org/index.php?title=Main_Page (visited on 06/06/2023).

[33] “Tensorflow lite micro library for arduino.” (), [Online]. Available: https://github.com/tensorflow/tflite-micro-arduino-examples (visited on 06/06/2023).



Background information

A.1. The Teensy 4.1 Microcontroller

In this section, we summarize the features of the Teensy 4.1 [9] microcontroller board. We discuss its hardware specifications, computational capacity, power requirements, and the features that make it suitable for the implementation of AI algorithms. The constraints posed by this device and the opportunities will also be discussed.

The Teensy 4.1 is a compact, versatile microcontroller board developed by PJRC. As an advanced yet affordable piece of hardware, it is a suitable platform for implementing AI algorithms.

A.1.1. Hardware Specifications

- ARM Cortex-M7 processor operating at 600 MHz (with cooling up to 1 GHz). It uses a Dual Issue Superscalar Architecture, allowing it to achieve 2 instructions per cycle about 40% to 50% of the time.
- It offers two 512Kb RAM chips, the first synchronous with the clock, the second at 1/4 clock speed, making the second chip non-ideal for storing of large matrices.
- 2 MB of on-board flash.
- The Floating Point Unit (FPU) performs 32-bit float and 64-bit double precision math in hardware, 32-bit float speed is approximately the same speed as integer math. The training of AI models requires a lot of 32-bit float operations, making the FPU a useful addition.
- Micro SD card reader. This is useful for storing datasets.

A.1.2. Constraints and Opportunities

Despite its advantages, the Teensy 4.1 presents some constraints when used for AI applications. The available RAM and flash memory, although relatively large for a microcontroller, limits us to small models. Additionally, the absence of a dedicated hardware accelerator for AI tasks means all computations need to be handled by the main processor. Dedicated hardware like the Neural Processing Unit present on the NXP MCX N94 can significantly improve power utilization and computation speed.

The Teensy 4.1, despite some limitations, offers a promising platform for implementing AI algorithms. Its hardware capabilities, combined with its affordability, present an unique platform for AI on the edge.

A.2. Fashion MNIST Dataset

The Fashion MNIST dataset, introduced by Zalando [10], serves as a drop-in replacement for the original MNIST dataset, which has become a standard benchmark in the field of machine learning. Although the original MNIST dataset of handwritten digits is very useful, it is often considered too

simple and overused. The Fashion MNIST dataset was created to address these issues, introducing more complexity while maintaining the same format as the original MNIST.

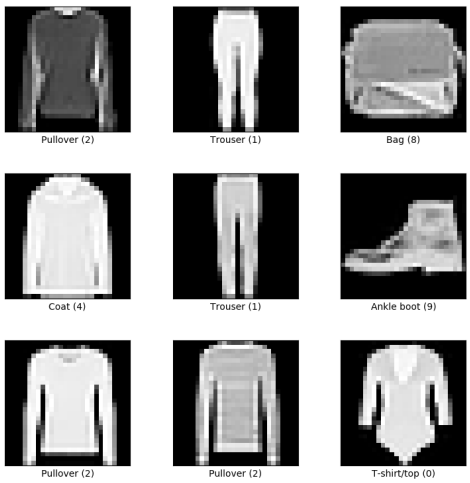


Figure A.1: Example images of the Fashion MNIST dataset. Figure obtained from [31]

A.2.1. Composition of the Dataset

The Fashion MNIST dataset consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 different classes, which represent different clothing items: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot.

A.2.2. Challenges Presented by the Dataset

While the Fashion MNIST is more complex than the original MNIST, it is still a relatively simple dataset compared to many real-world image recognition tasks. However, the diversity of clothing items, the variations in their appearances introduce a higher level of complexity compared to recognizing simple digits, as can be observed from a PyMDE comparison in Figure A.4, making the dataset more challenging than MNIST for benchmarking and validating AI algorithms.

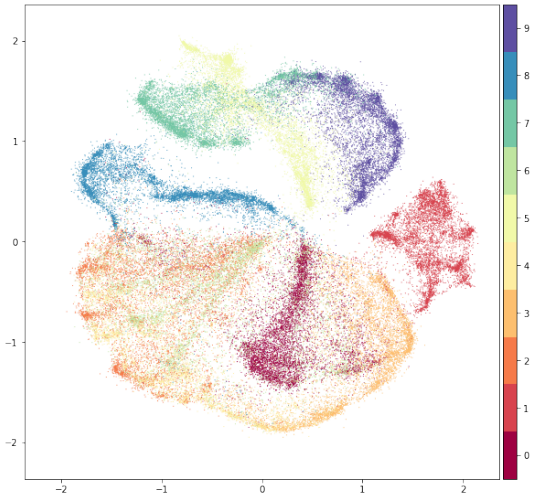


Figure A.2: Fashion MNIST

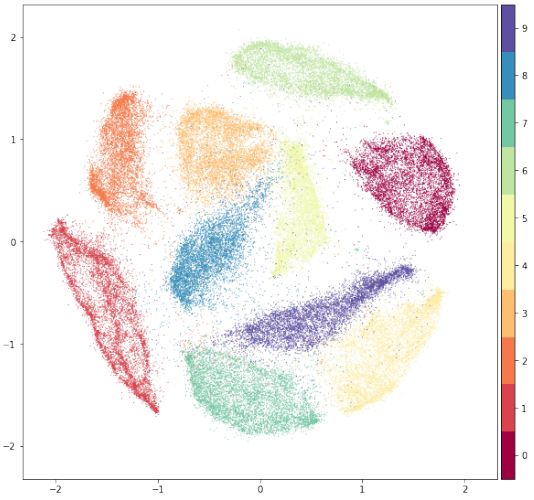


Figure A.3: MNIST

Figure A.4: PyMDE feature complexity comparison between Fashion MNIST and regular MNIST

A.2.3. Appropriateness for the Study

The choice of the Fashion MNIST dataset for this study is appropriate for several reasons. First, its compatibility with the original MNIST dataset means that it can be used with minimal changes to any learning algorithm train on the MNIST dataset. Second, it provides a more demanding test for the Backpropagation and Forward-Forward algorithms compared to the original MNIST set, yet remains manageable for a microcontroller like the Teensy 4.1. Lastly, using a well-known and commonly used dataset like the Fashion MNIST allows for easy comparison with other studies and models.

B

Results

B.1. Backpropagation

B.1.1. Baseline

Table B.1: Hyperparameters for the baseline configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	0	0.01	0	0

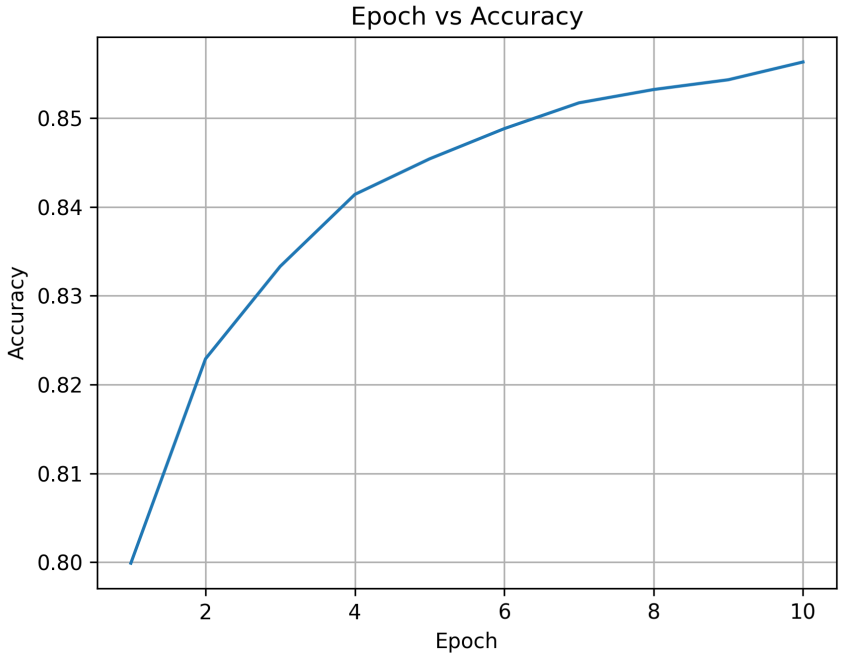


Figure B.1: Accuracy of Baseline over 10 epochs

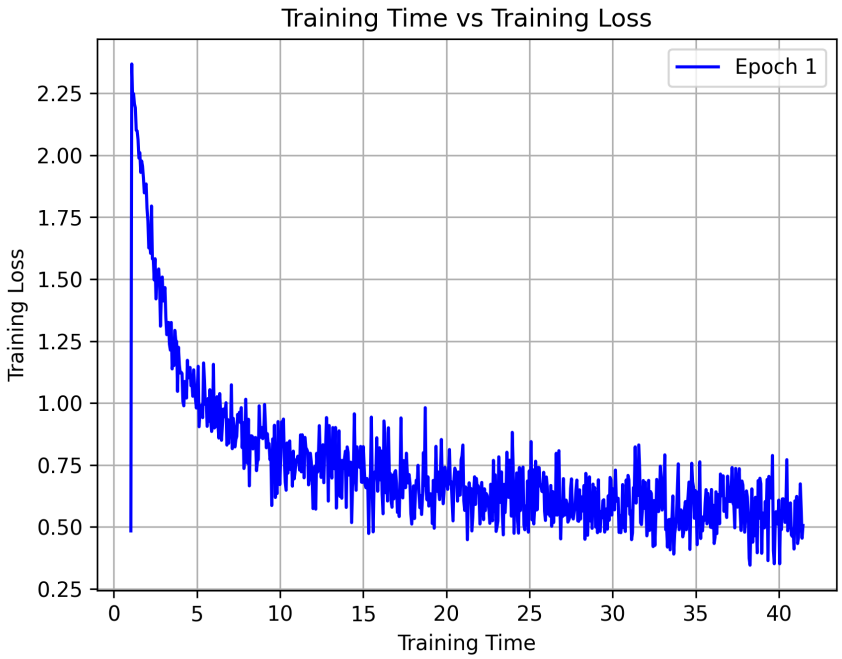


Figure B.2: Loss of Baseline during the first epoch (Loss vs Time(s))

Epoch	Accuracy	Test Time (s)
1	0.7999	4.162
2	0.8229	4.162
3	0.8333	4.163
4	0.8414	4.162
5	0.8454	4.162
6	0.8488	4.162
7	0.8517	4.161
8	0.8532	4.162
9	0.8543	4.162
10	0.8563	4.162

Table B.2: Test accuracy and time per epoch for the baseline configuration.

B.1.2. Single-layer with LR decay

Table B.3: Hyperparameters for the single-layer with LR decay configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	0	0.1	0.95	0

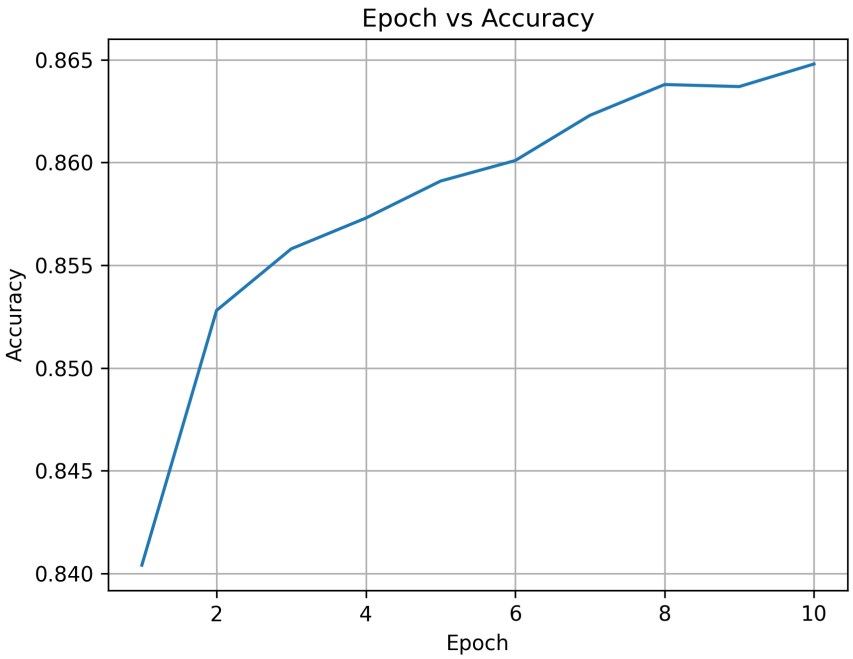


Figure B.3: Accuracy of the learning rate decay configuration over 10 epochs

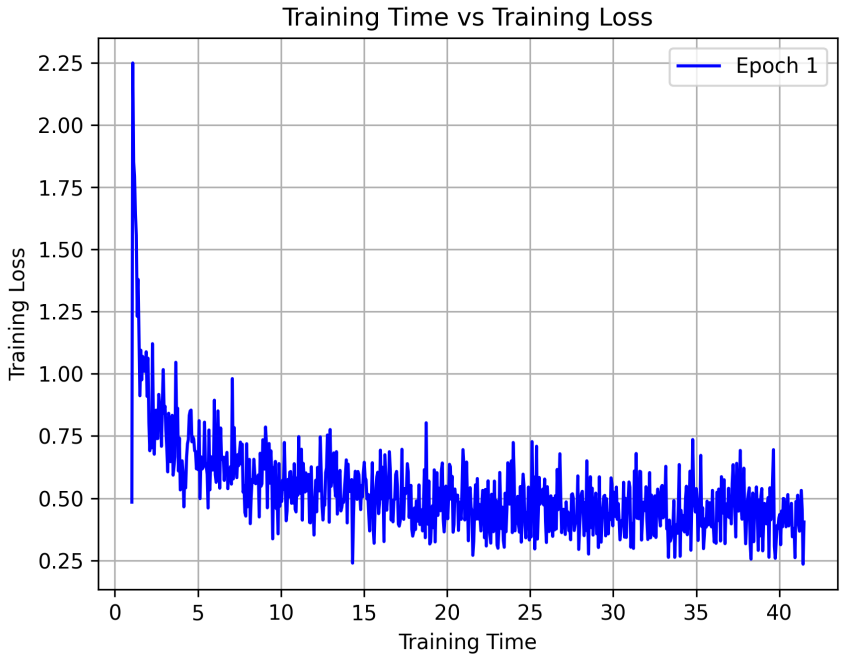


Figure B.4: Loss of the learning rate decay configuration during first epoch (Loss vs Time(s))

Epoch	Accuracy	Test Time (s)
1	0.8404	4.1618
2	0.8528	4.1607
3	0.8558	4.1618
4	0.8573	4.1619
5	0.8591	4.1610
6	0.8601	4.1610
7	0.8623	4.1614
8	0.8638	4.1605
9	0.8637	4.1612
10	0.8648	4.1613

Table B.4: Test accuracy and time per epoch for the single-layer with LR decay configuration.

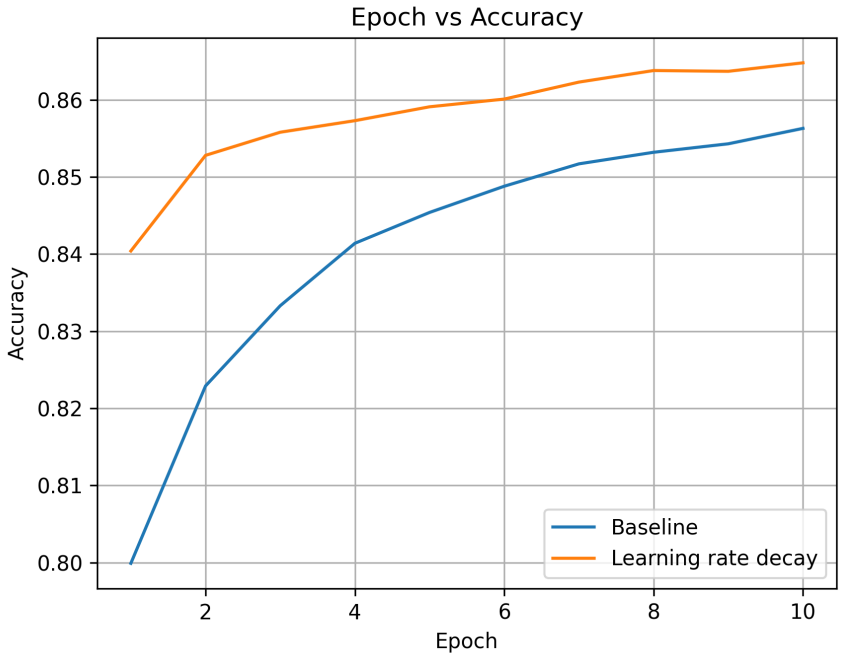


Figure B.5: Accuracy of the Baseline vs the LR decay configuration over 10 epochs

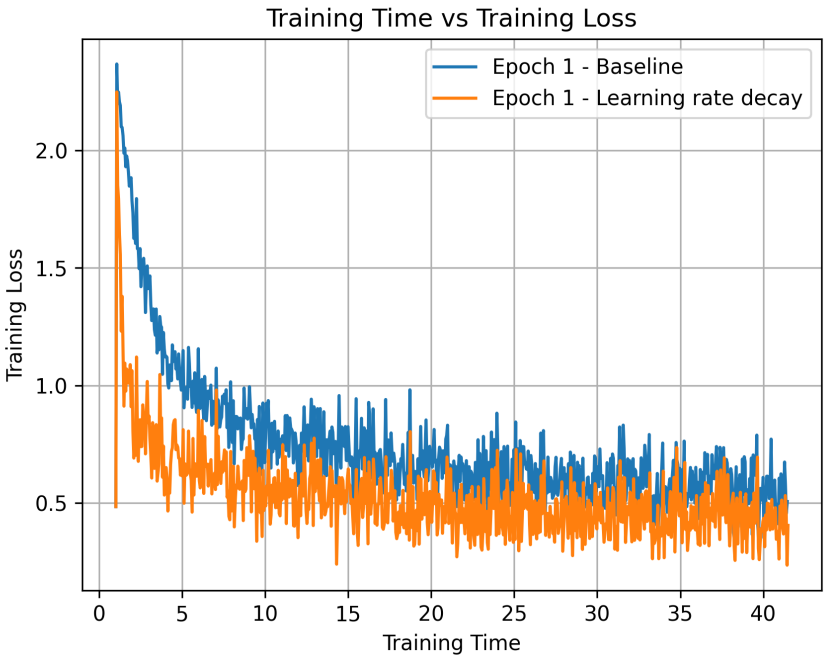


Figure B.6: Loss of the Baseline vs the LR decay configuration during the first epoch (Loss vs Time(s))

B.1.3. Single-layer with momentum

Table B.5: Hyperparameters for the single-layer with momentum configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	0	0.01	0	0.9

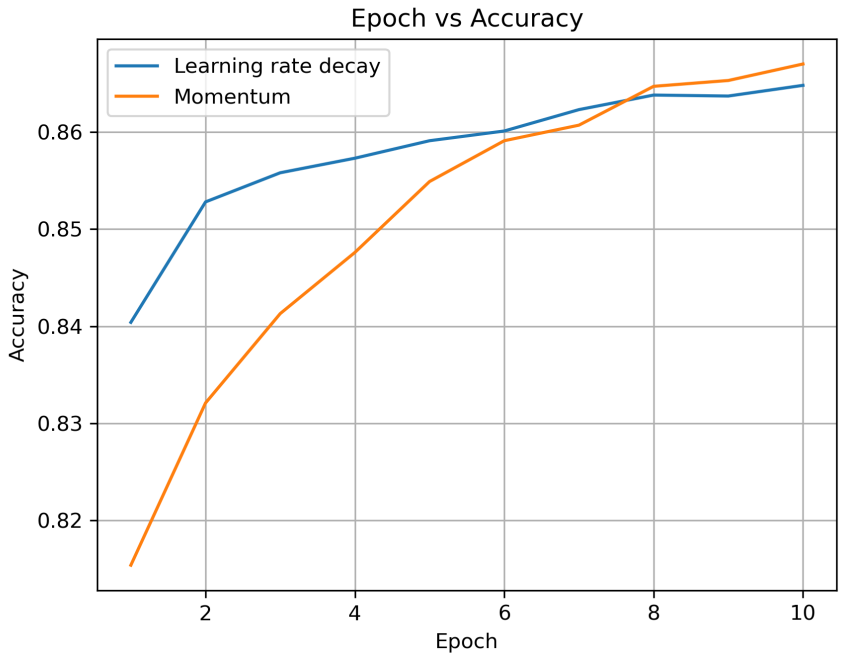


Figure B.7: Accuracy of training with Learning rate decay vs Momentum configuration over 10 epochs

B.1.4. Dual-layer

Table B.6: Hyperparameters for the baseline configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	32	0.01	0	0

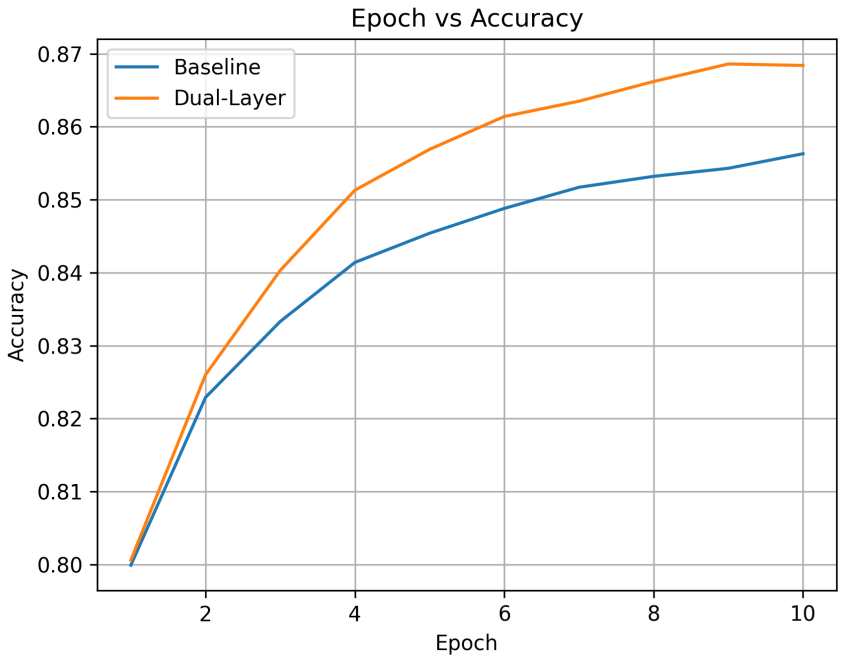


Figure B.8: Accuracy of the Baseline vs Dual-Layer configuration over 10 epochs

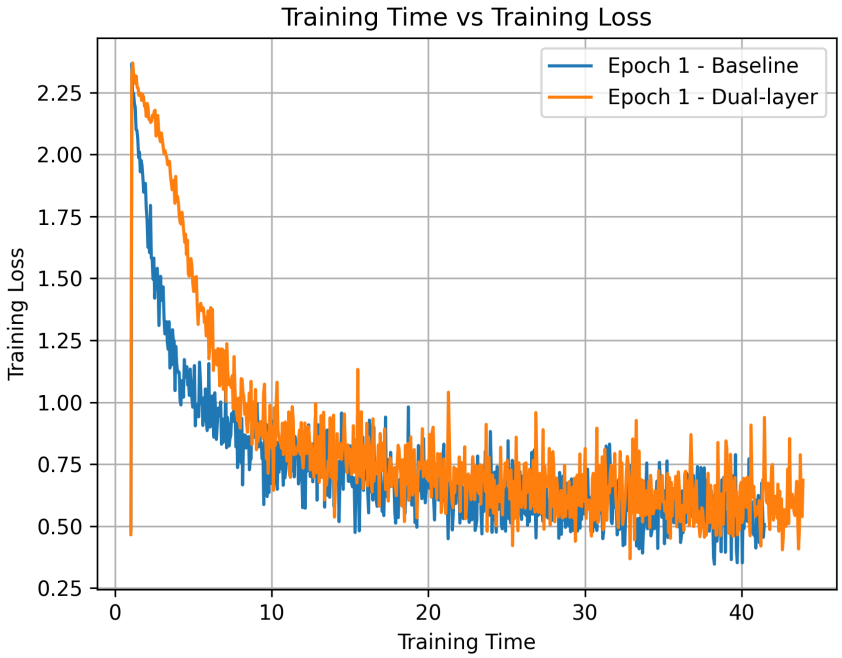


Figure B.9: Loss of the Baseline vs Dual-Layer configuration during the first epoch (Loss vs Time(s))

B.1.5. Dual-layer with LR decay

Table B.7: Hyperparameters for the baseline configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	32	0.1	0.95	0

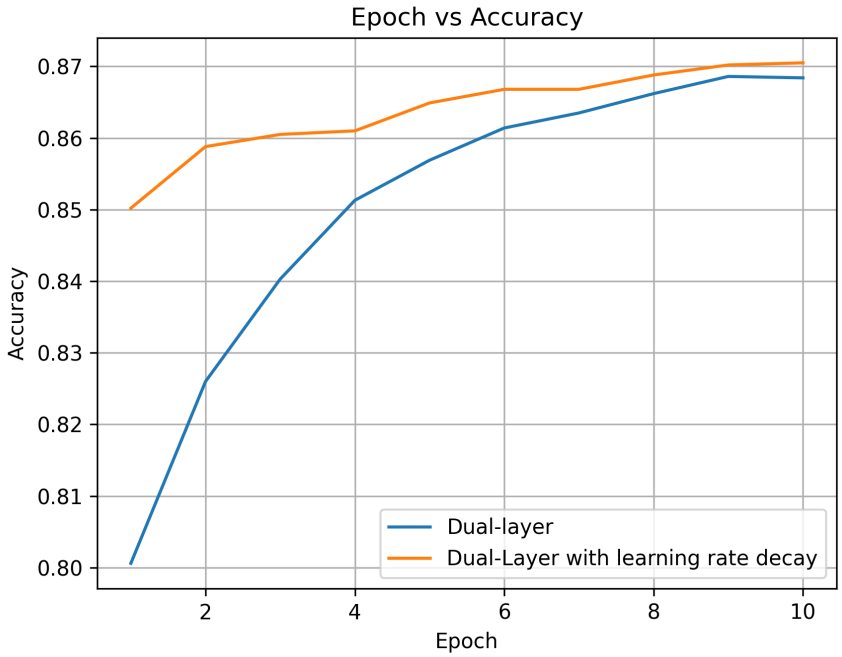


Figure B.10: Accuracy of Dual-layer without learning rate decay vs Dual-layer with learning rate decay configuration over 10 epochs

B.1.6. Larger networks using memory-optimized SGD

Table B.8: Hyperparameters for the baseline configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
80	80	0.1	0.95	0

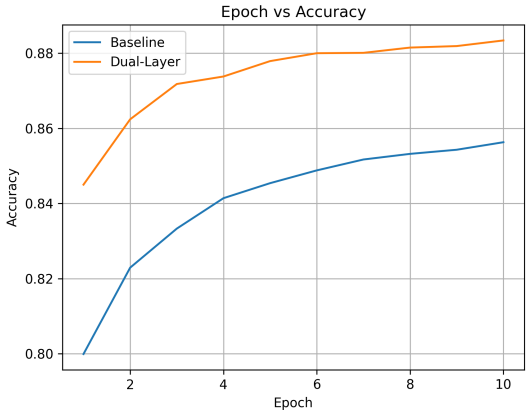


Figure B.11: Accuracy of Baseline vs Maximum Dual-Layer configuration over 10 epochs

B.2. Forward-Forward

B.2.1. Baseline

Table B.9: Hyperparameters for the baseline configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	0	0.01	0	0

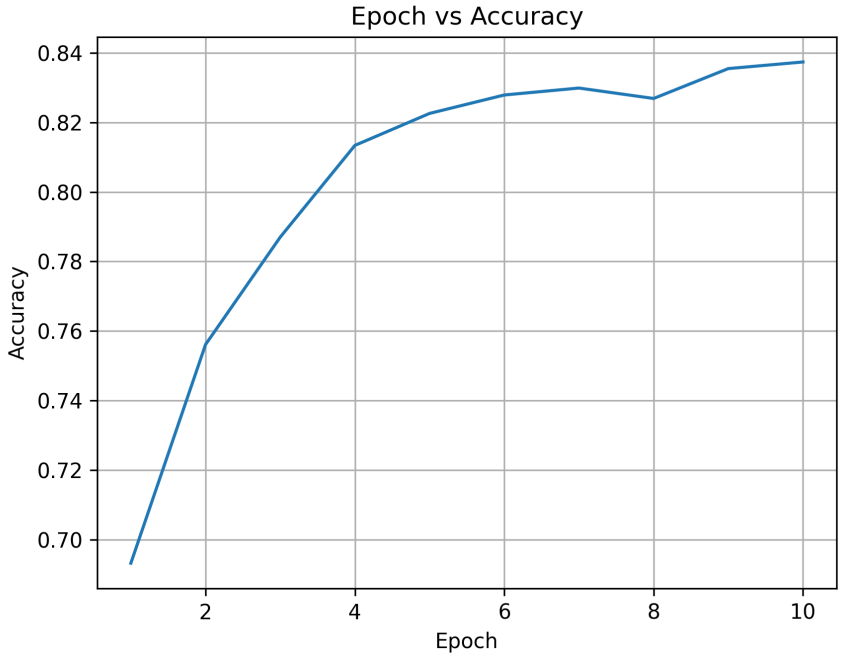


Figure B.12: Accuracy of Baseline over 10 epochs

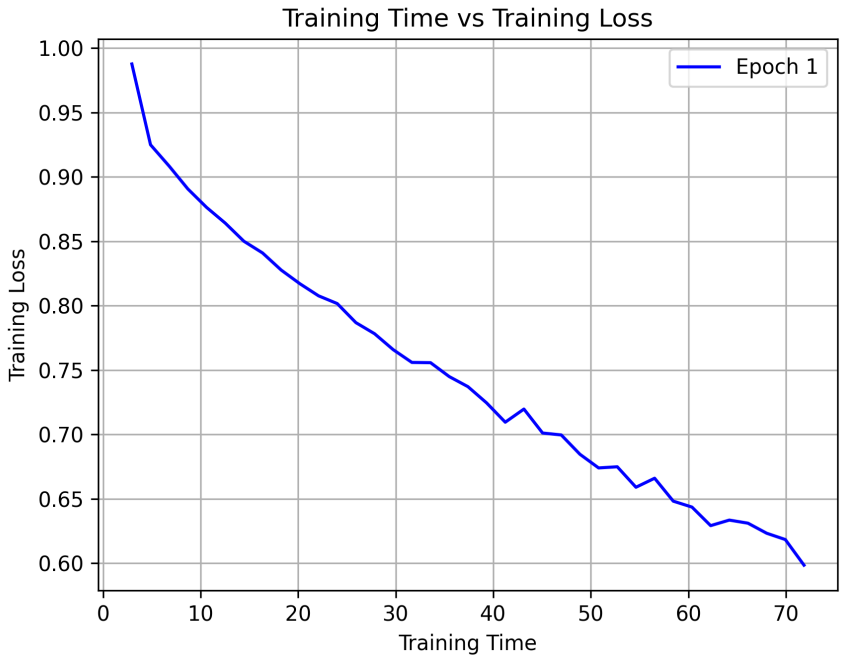


Figure B.13: Loss of the Baseline vs the LR decay configuration during the first epoch (Loss vs Time(s))

B.2.2. Dual-Layer Network

Table B.10: Hyperparameters for the baseline configuration

Neurons first layer	Neurons second layer	Initial LR	LR decay factor	Momentum
32	32	0.01	0	0

B.3. Comparison with other groups

In this section, the results of the microcontroller team are presented along with the results of the software and FPGA teams.

B.3.1. BP Baseline

Table B.11: Comparison table of the backpropagation baseline model (784-32-10, no lr decay, no momentum) between the different groups. From this table, it can be seen that the model trained with on-device backpropagation and SGD performs similarly to the model trained using TensorFlow. The quantized model also performs similarly to the models running on the PC and FPGA. Hyperparameters used for training: Batch size = 16, lr = 0.01, momentum = 0, no lr decay

Metrics	**TF (PC)**	**TF Lite (PC)**	**TF Lite (MCU)**	**Local MCU**	**FPGA**
Datatype	fp32	int8	int8	fp32	int8
Accuracy on test set after 1 epochs	80.72	NA	NA	79.99	NA
Accuracy on test set after 5 epochs	85	NA	NA	84.54	NA
Accuracy on test set after 10 epochs	85.82	NA	85.6	85.63	NA
Accuracy on test set after 30 epochs	86.41	86.33	86.33	NA	85.53
Memory required (KiB) (Variable | Code)	NA	NA	60.16 | 105.74	265.22 | 99.90	25.08 | 1.34
Inference time per sample(us)	65.02	34.62	70.19	258.4	14.38
Training time (s)	142.2	NA	NA	372.9	NA
Training time per epoch (s)	4.74	NA	NA	37.29	NA

B.3.2. BP Baseline + LR decay

Table B.12: Comparison table of the backpropagation baseline model (784-32-10, lr decay, no momentum) between the different groups. From this table, it can be seen that the model trained with on-device backpropagation and SGD performs similarly to the model trained using TensorFlow. The quantized model also performs similarly to the models running on the PC. Hyperparameters used for training: Batch size = 16, lr = 0.01, momentum = 0, lr decay = 0.95 every 200 batches

Metric	TF (PC)	TF Lite (PC)	TF Lite (MCU)	Local MCU
Datatype	fp32	int8	int8	fp32
Accuracy on test set after 1 epoch	84.17	–	–	84.04
Accuracy on test set after 5 epoch	85.8	–	–	85.91
Accuracy on test set after 10 epochs	86.23	–	–	86.48
Epochs trained	26	–	–	10
Accuracy for given epochs	86.74	86.68	86.68	–
Memory required (KiB) (Variable | Code)	–	25.08 | 1.34	60.16|105.74	265.22 | 99.90
Inference time per sample(us)	86.97	30.55	70.1	258.4
Training time (s)	262.6	–	–	372.9
Training time per epoch (s)	10.1	–	–	37.29
Energy / inference (mJ)	–	–	0.046	0.199

B.3.3. BP Baseline + Momentum

Table B.13: Comparison table of the backpropagation baseline model (784-32-10, no lr decay, momentum) between the different groups. From this table, it can be seen that the model trained with on-device backpropagation and SGD performs similarly to the model trained using TensorFlow. The quantized model also performs similarly to the models running on the PC. Hyperparameters used for training: Batch size = 16, lr = 0.01, momentum = 0.9, no lr decay

Metric	TF (PC)	TF Lite (PC)	TF Lite (MCU)	Local MCU
Datatype	fp32	int8	int8	fp32
Accuracy on test set after 1 epoch	83.05	–	–	81.54
Accuracy on test set after 5 epoch	85.95	–	–	85.49
Accuracy on test set after 10 epochs	86.94	–	–	86.7
Epochs trained	12	–	–	10
Accuracy for given epochs	86.43	86.45	86.43	–
Memory required (KiB) (Variable | Code)	–	25.08 | 1.34	60.16|105.74	265.22 | 100.52
Inference time per sample(us)	69.1	24.83	70.35	258.3
Training time (s)	82.88	–	–	378.6
Training time per epoch (s)	6.91	–	–	37.86
Energy / inference (mJ)	–	–	0.047	0.171

B.3.4. Dual layer BP

Table B.14: Comparison table of the backpropagation dual layer model (784-32-32-10, no lr decay, no momentum) between the different groups. From this table, it can be seen that the model trained with on-device backpropagation and SGD performs similarly to the model trained using TensorFlow. The quantized model also performs similarly to the models running on the PC and FPGA. Hyperparameters used for training: Batch size = 16, lr = 0.01, momentum = 0

Metric	TF (PC)	TF Lite (PC)	TF Lite (MCU)	Local MCU	FPGA
Datatype	fp32	int8	int8	fp32	int8
Accuracy on test set after 1 epoch	81.03	–	–	80.06	–
Accuracy on test set after 5 epoch	85.07	–	–	85.69	–
Accuracy on test set after 10 epochs	86.66	–	–	86.84	–
Epochs trained	22	–	–	10	–
Accuracy for given epochs	87.46	87.36	87.36	–	85.67
Memory required (KiB) (Variable | Code)	–	26.20 | 2.33	61.16 | 105.74	278.47 | 102.02	–
Inference time per sample(us)	72.34	37.47	75.2	270.04	15.24
Training time (s)	123.9	–	–	396.6	–
Training time per epoch (s)	5.63	–	–	39.66	–
Energy / inference (mJ)	–	–	0.05	0.179	0.029

B.3.5. Baseline FF

Table B.15: Comparison table of the forward-forward baseline model (784-32, no lr decay, no momentum, batch size = 16, lr = 0.1) between the software and microcontroller groups. Accuracies at different training stages are similar between models trained on the MCU and models trained using TensorFlow.

Metrics	TF (PC)	Local MCU
Datatype	fp32	fp32
Accuracy on test set after 1 epochs	68.72	69.32
Accuracy on test set after 5 epochs	79.50	82.26
Accuracy on test set after 10 epochs	82.55	83.74
Memory required (KiB) (Variable | Code)	NA	318.19 | 98.09
Inference time per sample(us)	604.1	2533.27
Training time per epoch (s)	29.24	71.18

Table B.16: Comparison table of the forward-forward baseline model (784-32, no lr decay, no momentum, batch size = 32, lr = 0.1) between the software and microcontroller groups using TensorFlow Lite. A significant drop in accuracy can be seen when going from full-precision to quantized.

Metrics	TF (PC)	TF Lite (PC)	TF Lite (MCU)
Datatype	fp32	int8	int8
Accuracy on test set after 20 epochs	82.43	68.9	69.66
Memory required (KiB) (Variable | Code)	NA	25.32|1.48	57.06 | 105.18
Inference time per sample(us)	604.1	8771.5	1065.4
Training time per epoch (s)	29.24	NA	NA

B.3.6. Dual Layer FF

Table B.17: Comparison table of the forward-forward dual layer model (784-32-32, no lr decay, no momentum, batch size = 32, lr = 0.1) between the software and microcontroller groups using TensorFlow Lite. A significant drop in accuracy can be seen when going from full-precision to quantized.

Metrics	TF (PC)	TF Lite (PC)	TF Lite (MCU)	Local (MCU)
Datatype	fp32	int8	int8	fp32
Accuracy on test set after 10 epochs	79.46	59.79	67.93	81.59
Memory required (KiB) (Variable | Code)	NA	26.58|2.96	70.47 | 105.18	327.31 | 99.21
Inference time per sample(us)	1374	16065	1189.1	2794.27
Training time per epoch (s)	57.006	NA	NA	106.3

B.4. Forward-Forward with batch size of 32

Table B.18: Forward-Forward Model Comparison. Memory usage between parentheses is using the memory-optimized version of SGD. Batch size = 32, network topologies and other hyperparameters are unchanged with respect to chapter 5

Metrics	Baseline	FF Dual-layer
Accuracy on test set after 1 epoch	0.6202	0.5952
Accuracy on test set after 5 epochs	0.7668	0.7942
Accuracy on test set after 10 epochs	0.809	0.8121
Memory required (kB) Variable	318.19 (220.06)	327.32 (229.19)
Memory required (kB) Code	98.09	99.21
Inference time (us) / image	2533.3	2794.3
Training time (s) / epoch	71.18	106.30
Energy (mJ) / inference	1.676	1.848

B.5. Direct comparison for foward-forward and backpropagation

Table B.19: Direct comparison between models trained using backpropagation and forward-forward. The used model topologies are the same as before (784-32-10, 784-32-32-10 for BP, 784-32, 784-32-32 for FF). No LR decay or momentum are used during training.

Metrics	BP Single hidden layer	BP Dual hidden layer	FF Single-layer	FF Dual-layer
Accuracy on test set after 1 epoch	0.7999	0.8006	0.6932	0.7002
Accuracy on test set after 5 epochs	0.8454	0.8569	0.8226	0.8193
Accuracy on test set after 10 epochs	0.8563	0.8684	0.8374	0.8206
Memory required (kB) Variable	265.22 (165.81)	278.47 (173.94)	267.06 (168.94)	274.19 (176.06)
Memory required (kB) Code	99.90	102.02	98.09	99.21
Inference time (us) / image	258.4	270.0	2533.3	2794.3
Training time (s) / epoch	37.29	39.66	71.18	106.30
Energy (mJ) / inference	0.171	0.179	1.676	1.848

B.6. Justification for testing methodology

Below, the accuracy vs epochs is plotted for a dual layer model trained using TensorFlow with momentum and enabled. This plot is included to elucidate the decision to not include accuracies of models trained for more than 10 epochs in the results. It becomes clear that most training happens in epochs 1-10. While this proof is anecdotal and there are certainly networks that benefit from training more than 10 epochs, it is reasonable to think that it is applicable to the networks trained on the microcontroller. For example, the network topology in this case is the same, and so is the optimizer. Training

networks on the microcontroller also takes a considerable amount of time (especially forward-forward, which would take approximately 1.5 hours to train with 50 epochs). This is why the decision was made to only report accuracies up to 10 epochs.

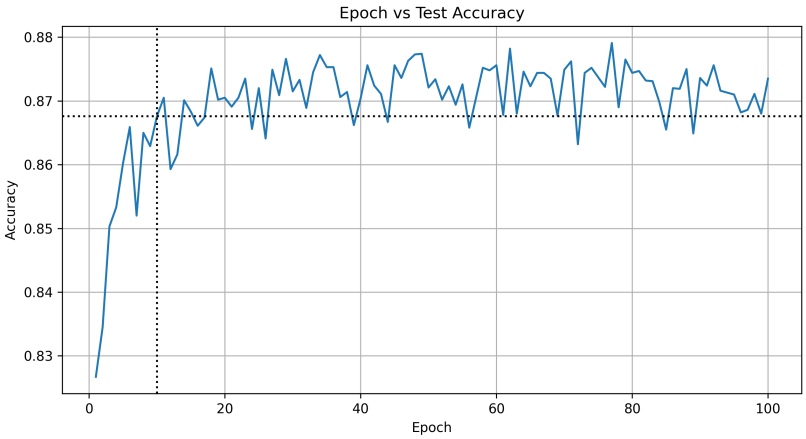


Figure B.14: Test accuracy on fashion MNIST using a dual hidden layer model (784-32-32-10) trained with SGD (lr=0.001, momentum=0.9) using TensorFlow. This plot shows that most of the training occurs in the first 10 epochs.

C

General code / support structure

There are pieces of code that are generic and can be used with all algorithms. Namely, the data loading code handles loading the train and test datasets into memory, the interfacing code provides a user interface over the serial connection, and the linear algebra library provides all functionality regarding vector and matrix operations.

C.1. Data Loading

As described in appendix A, the Teensy 4.1 microcontroller has 2 MB of on-board flash. This makes it impossible to load the entire fashion MNIST dataset onto flash (the fashion MNIST dataset is 26 MB[10]). This means that the dataset must be loaded into memory in mini-batches. There are two main ways to achieve this: over a serial connection or by reading from an SD card. Loading from the SD card is chosen because it is easy to implement. First, the data is read from a CSV file, because this allowed us to put the dataset in a human-readable format onto the SD card. We choose to pre-flatten the images, because this would be easier to read into memory and store in files. Quickly, it became apparent that the string processing on the Teensy was massively bottlenecking the speed at which new examples could be loaded into memory. To remedy this, a Python script was created to put the entire dataset into a binary format. This is easy in the case of images, because all pixel values are represented by a single byte. Then, on the Teensy, the contents of the binary file can be copied directly into memory, avoiding the need to do computationally expensive string processing. This massively reduced the amount of time it took to load data from the SD card into memory, however, loading data from the SD card still takes a considerable amount of time when compared to running computations for the neural network inference and training. The amount of data that was loaded from the SD card at a time was varied. Two configurations were investigated: loading per example and per batch. There was not a significant difference in data loading times between these configurations. Ultimately, the per-example loading was used, because this requires a smaller read-buffer, thus saving memory. The normalization of the input data is done on-device, as we think this is more representative of the real world use cases of both inference-only and on-device training.

C.2. Interfacing with MCU

In order to run experiments, a simple user interface was built to interface with the Teensy over a serial connection with an attached computer. This user interface allows training for multiple epochs and testing the models, as well as saving the trained weight matrices and bias vectors to a human readable format. A tool for plotting loss graphs was built in Python which uses the serial connection to obtain real-time running losses from the microcontroller.

C.3. Linear Algebra Library

For both training and inferring neural networks, vector and matrix operations are paramount. There exist linear algebra libraries written in c++ [32], however, these libraries are often large and would take up a significant amount of memory. Therefore, a custom, lightweight, linear algebra library was written

that only contains the required operations for training and inferring fully connected neural networks. By writing a custom linear algebra library, it is also possible to make optimizations to certain operations. The library works with both statically and dynamically allocated matrices and vectors. Each matrix or vector is a C-struct that consists of a pointer to a storage array and metadata about how the array is organized, such as vector length, and the amount of rows and columns. This makes re-using allocated memory easy. Instead of having to create a new matrix or vector, one can simply alter the metadata of the struct to change dimensions. The library also contains various combined operations. For example, matrix multiplication and transposition can be done in a single function call, removing the need to allocate a second matrix to store the result of the transposition.

C.4. TensorFlow Lite Micro

TensorFlow Lite Micro is not officially supported on the Teensy 4.1. This means that, in order to make use of it, the library has to be ported to the Teensy. This is accomplished by adapting the TF Lite Micro library that was developed for the Arduino Nano 33 BLE [33] to work with the Teensy 4.1. This is done because the Arduino board has a microcontroller with the same core architecture as the Teensy (namely, the Arduino has an ARM cortex M4 based MCU and the Teensy has an ARM cortex M7 based MCU).

D

Mathematical definition loss and gradient

D.1. Forward-Forward contrastive loss

In this section, the mathematical definition and the gradient with respect to layer weights and biases of the contrastive loss function of forward-forward is given. Matrices are denoted by **bold** letters, vectors are denoted with $\vec{arrows}$. Operations (squaring, addition) of a matrix / vector with a scalar is element-wise). Furthermore, sums of matrices can be taken over a certain axis (axis 0 = rows, axis 1 = columns). Example of summing over the row dimension:

$$\sum_{axis=0} \mathbf{Y}$$

The operation above would result in a vector with a length equal to the column dimension of the matrix $\mathbf{Y}$.

D.1.1. Definition

The definition of the loss follows [30]. It is given for positive data. The following symbol definitions are used:

- f: amount input features.
- n: batch size.
- h: amount of neurons in the layer.
- θ: threshold hyperparameter of the loss.
- $\mathcal{L}$: loss.
- $\vec{b}$: bias vector (of length h).
- $\vec{g}$: goodness vector (length n), which stores the mean of square activations per example.
- $\mathbf{X}$: batch matrix (size (n, f))
- $\mathbf{W}$: the weight matrix (size (f, h))
- $\mathbf{Y}'$: pre-activation layer outputs (size (n, h)).
- $\mathbf{Y}$: layer activations after applying the ReLU activation (size (n, h)).

The loss $\mathcal{L}$ is defined as follows:

$$\mathbf{Y}' = \mathbf{X} \times \mathbf{W} + \vec{b}$$
$$\mathbf{Y} = \max(0, \mathbf{Y}')$$
$$\vec{g} = \frac{1}{h} \sum_{axis=1} \mathbf{Y}^2$$
$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} ln(1 + e^{-g_i + \theta})$$

D.1.2. Gradient

Now, the gradient with respect to the layer weights and biases is calculated using the chain rule. To make the derivation easier, the following additional symbols are defined:

- $\alpha = -g_i + \theta$

The gradient is then computed as follows:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \frac{1}{n + ne^{-\alpha}}$$
$$\frac{\partial \alpha}{\partial g_i} = -1$$
$$\frac{\partial g_i}{\partial Y_{ij}} = \frac{2}{h} Y_{ij}$$
$$\frac{\partial Y_{ij}}{\partial Y'_{ij}} = \begin{cases} 1 & \text{if } Y'_{ij} \geq 0 \\ 0 & \text{if } Y'_{ij} < 0 \end{cases}$$

Now, the chain rule is applied. Because $Y_{ij} = Y'_{ij}$ if $Y'_{ij} \geq 0$ and $Y_{ij} = 0$ if $Y'_{ij} < 0$, multiplying by the last partial derivative does not change the derivative of $\mathcal{L}$ with respect to Y'_{ij}. The partial derivatives are now combined:

$$\frac{\partial \mathcal{L}}{\partial Y'_{ij}} = -\frac{1}{n + ne^{g_i - \theta}} \frac{2}{h} Y_{ij}$$

Now, a gradient matrix $\mathbf{Y}'_{grad}$ is defined:

$$\mathbf{Y}'_{grad} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial Y'_{00}} & \cdots & \frac{\partial \mathcal{L}}{\partial Y'_{0h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial Y'_{n0}} & \cdots & \frac{\partial \mathcal{L}}{\partial Y'_{nh}} \end{bmatrix}$$

The gradients of the loss with respect to the weights and biases can now be computed:

$$\mathbf{W}_{grad} = \mathbf{X}^T \times \mathbf{Y}'_{grad}$$
$$\vec{b}_{grad} = \sum_{axis=0} \mathbf{Y}'_{grad}$$

For negative data, $-\alpha$ is used instead of α. The gradient derivation for this is now trivial.

– The End –