AI on Low-Cost Hardware

Software Subgroup

Bsc Thesis

Christian van den Berg
Hong Jie Zheng

$$\vec{\nabla} \mathcal{L}$$

AI on Low-Cost Hardware

Software Subgroup

by

Christian van den Berg
Hong Jie Zheng

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Wednesday June 21st, 2023 at 13:00.

Student number:	Christian van den Berg	5401674
	Hong Jie Zheng	5135826
Project duration:	April 24, 2023 – June 15, 2023	
Thesis committee:	Dr. Justin Dauwels	
	Dr. Charlotte Frenkel	
	Dr. Ir. Rob Remis	
	Dr. Sebastian Feld	

TUDelft

Abstract

Artificial Intelligence has become a dominant part of our lives, however, complex artificial intelligence models tend to use a lot of energy, computationally complex operations, and a lot of memory resources. Therefore, it excluded a whole class of hardware in its applicability. Namely, relatively resource-constrained low-cost hardware. This paper investigates learning methods that are potentially better suited for these types of devices: the forward-forward algorithm and Hebbian learning rules. The results are compared to backpropagation with equivalent network configurations, training hyperparameters and internal data types on different types of low-cost hardware. Backpropagation has consistently outperformed other algorithms in various tests. It exhibits higher accuracy, faster training, and faster inference compared to forward-forward models. However, forward-forward models can come close to matching backpropagation's performance, but they suffer from longer training times and decreased performance with multi-layer networks. Additionally, a poorly trained forward-forward model is sensitive to quantization, resulting in a significant drop in accuracy. On the other hand, forward-forward models offer the benefit of independently training each layer, allowing for more flexibility in optimizing the training process. Hebbian models were not found to be competitive, displaying performance below the required threshold. Smaller models for MCU and FPGA would likely perform even worse.

Preface

This thesis is a component of the Bachelor Graduation Project. As members of the Software group, the project objective was to research and implement the forward-forward algorithm and Hebbian learning rules on relatively resource-constrained hardware, with a focus on high-level frameworks like TensorFlow and TensorFlow Lite. The translation from the high-level framework provided by TensorFlow to the low-level implementation on hardware proved to pose a significant challenge. Not only because of the layers under the surface of TensorFlow, TensorFlow itself also showed some inflexibility we had to work around. Nonetheless the work done was very insightful for us. AI still has a lot left to discover.
We would like to express gratitude to our supervisors, dr. Charlotte Frenkel, dr.ir. Justin Dauwels and Prof. Dr. Frans Widdershoven for the guidance during the project and for the exposure to a vast range of interesting concepts while doing the project. In addition we would like to thank Yarib Nevarez Esparza, a colleague of Prof. Dr. Frans Widdershoven, who had also taken up the role of supervisor. Finally, we would like to thank our colleagues Marijn Adriaanse, Li Ou Hu from the FPGA group and Jarl Brand, Mano Rom from the MCU group for their effort in this project and the enjoyable collaboration.

Christian van den Berg
Hong Jie Zheng
Delft, July 2023

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DL	Deep Learning
FF	Forward-Forward Algorithm
FPGA	Field-programmable gate array
GPU	Graphics Processing Unit
IoT	Internet of Things
MCU	Microcontroller Unit
ML	Machine Learning
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
STDP	Spike Time Dependent Plasticity
GAN	Generative Adversarial Network
LSTM	Long Short-Term Memory

Symbols

Symbol	Definition
$\mathcal{L}$	Loss function
ϕ	Forward-forward loss input
θ	Activation threshold hyperparameter
η	Learning rate hyperparameter of SGD
μ	Learning rate decay hyperparameter
B	Set of batch samples with $\|B\|$ as the number of elements in set and $s \in B$ is an element of B
(L)	Layer index of NN. e.g. weight Matrix $\mathbf{W}^{(L)}$ of layer (L)
W	Weight matrix: W_{ij} (entry of Weight matrix)
a	Activation Vector: a_i (entry of Activation Vector)
z	Linear combination of the input and weights and biases: z_i (entry of Linear combination)
x	Input vector of NN: x_i (entry of Input vector)
y	Output vector of NN: y_i (entry of Output vector)
b	Bias vector: b_i (entry of Bias vector)
V	Velocity matrix: V_{ij} (entry of Velocity matrix)

1
Introduction

In this fast-paced and technologically driven era, Artificial Intelligence (AI) has emerged as a ground-breaking innovation that is transforming various aspects of our lives. Its wide-ranging applications and potential have made it an indispensable part of our daily lives, influencing numerous industries and sectors including healthcare, finance, transportation, education, and entertainment. As we navigate the complexities of the 21st century, understanding the relevance of AI is essential to grasp the profound impact it has on society, economy, and our future. The rapid advancements in computing power, availability of vast amounts of data, and breakthroughs in algorithm development have accelerated the progress of AI in recent years. AI systems are now capable of analyzing immense volumes of information, recognizing patterns, and making complex decisions with remarkable accuracy and efficiency.
The relevance of AI extends beyond high-end computing systems and expensive hardware. In recent years, there has been a significant effort to develop AI algorithms and techniques that can run effectively on low-cost hardware. This trend has improved access to AI technologies, allowing a broader range of individuals and organizations to leverage its benefits.
Low-cost hardware, such as single-board computers and edge devices, have become increasingly capable of performing AI tasks locally, making it possible to deploy AI applications directly on edge, without relying heavily on cloud infrastructure. This shift towards edge computing has numerous advantages, including reduced latency, improved privacy and security, and the ability to operate in environments with limited or intermittent internet connectivity. This is especially relevant for the Internet of Things (IoT) ecosystem [2]. As more devices become interconnected and generate vast amounts of data, running AI algorithms on the edge allows for real-time data processing and intelligent decision-making at the device level. This decentralized approach reduces the reliance on cloud infrastructure, minimizes network congestion, and enhances the overall responsiveness and efficiency of IoT systems. However, it is important to recognize the trade-offs associated with AI and hardware. Limited computational power and memory constraints impose limitations on the complexity and scale of AI applications. One way to tackle this problem is by optimizing algorithms. Backpropagation is a very popular training algorithm for machine learning, however it has some shortcomings.
In this paper two alternatives to backpropagation will be investigated, with a focus on their application on low-power devices, the forward-forward algorithm and Hebbian Learning. The forward-forward algorithm was proposed by Hinton [15] and is suggested to be better suited for low-power applications. Hebbian learning was proposed by Hebb [13]. This is based on the principle that neurons that fire together, wire together. These two methods are both local learning methods, the benefit of this is that it requires less computational resources than non-local methods, another benefit is that it is suggested that in cortex learning is closer to local learning methods.
The requirements for this project are specified in Chapter 2. Chapter 3 provides some background information on neural networks, backpropagation, and the shortcomings thereof. Chapter 4 and Chapter 5 describe the forward-forward algorithm and Hebbian learning respectively. The methodology of the research into the algorithms is laid out in Chapter 6. The results are shown in Chapter 7 and concluded in Chapter 8.

2

Programme of Requirements

The goal of this project is to investigate and implement local learning methods, such as forward-forward and Hebbian, on low-power hardware. The implementation on a physical microcontroller will be done by the MCU group The implementation on a FPGA will be done by the FPGA. Our goal is to research these algorithms and to assist the other groups in implementing these algorithms on their hardware.

2.1. Mandatory Requirements

The final product will be neural network models using local learning methods, these networks will have the following mandatory requirements:

2.1.1. Functional Requirements

- Implement neural networks using backpropagation
- Implement neural networks using local learning algorithms, the forward-forward algorithm or Hebbian learning
- Implement the neural networks using TensorFlow and Python
- Run the models on hardware using TensorFlow Lite
- The models must be able to fit in the memory of a Teensy 4.1 microcontroller
- A model must be provided which is able to run on a Teensy 4.1 microcontroller
- A model must be provided which is able to run on a Digilent ZedBoard FPGA

2.1.2. Non-Functional Requirements

- The accuracy of the models has to be at least 80% on the Fashion MNIST dataset
- The models have to fit on the microcontroller

2.2. Trade-off Requirements

For the trade-off requirements, we have the following in order of priority:

- Minimize the training time of the models
- Minimize the inference time of the models
- Maximize the test accuracy of the models on the Fashion MNIST dataset

Maximizing the accuracy of the models will often be in conflict with minimizing training time and inference time. Our goal is first to minimize the training time and inference time, since these affect the usability of the model the most. For some models, the accuracy will be maximized at the cost of training and inference time, but for most models the training and inference time will take priority. Furthermore, we want to have a wide variety of models to draw conclusions from. This allows us to draw conclusions about the scaling behaviour of the different learning methods and investigate the effects different hyperparameters have on the models.

3

Artificial Neural Networks

The basic building block of an Artificial Neural Network (ANN) is an artificial neuron, also called a perceptron, which takes in multiple inputs, applies a set of weights to the inputs, and passes the weighted sum through an activation function to produce an output. The activation function introduces non-linearity into the network, allowing it to learn complex patterns and make non-linear predictions. Artificial Neural networks are inspired by the brain, whether the human brain or the brain of another animal [8].
Neurons are organized into layers within a neural network. The simplest version of a neural network is the feedforward neural network (FNN), Fig. 3.1 shows such a network. The first layer is the input layer, which receives the initial data. The intermediate layers are called hidden layers, these layers are each assigned a set of weights which is applied on the input data. The final layer is the output layer, which produces the network's predictions or outputs. During training, the neural network learns to adjust the weights of its connections based on the input data and the desired outputs. This process is typically done using an optimization algorithm, such as gradient descent, to minimize a loss function that measures the discrepancy between the predicted outputs and the actual outputs. By iteratively adjusting the weights, the network gradually improves its ability to make accurate predictions.
Other neural networks include convolutional neural networks (CNN), recurrent neural networks (RNN), and more advanced architectures like long short-term memory (LSTM) networks and generative adversarial networks (GANs). Each type has its own structure and is suitable for different types of tasks. The focus in this paper will be on feedforward networks.

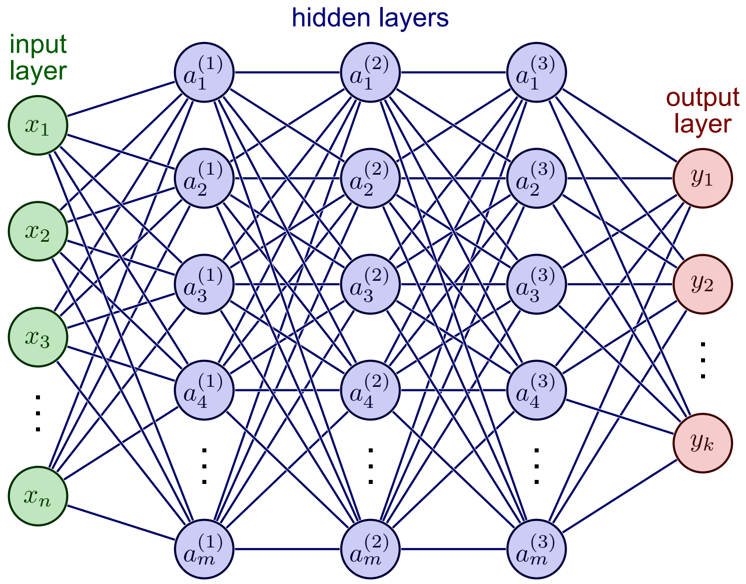


Figure 3.1: Typical artificial feedforward neural network. [23]

3.1. Backpropagation

Backpropagation or backprop is a supervised learning algorithm for feedforward artificial neural networks. The backpropagation algorithm consists of a forward pass and a backward pass. In the forward pass input data is fed into the network and propagated through the entire network, at the output layer a loss or cost function is applied. The loss indicates how far the network output is from the desired output. The operand used in calculating the relative errors back to the neuron weights is the gradient of the loss function. The gradients, 3.1, are used to update the weights for every training iteration. Consequently, a backward pass propagates back through the network and updates the weights with an optimizer. In the FNN shown in Fig. 3.1 the loss is determined at the output layer. The hidden layers contain the weights of the network. The weights of the neurons in these hidden layers are trained to minimize the loss at the output layer.

$$\vec{\nabla}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \ldots, \frac{\partial \mathcal{L}}{\partial w_n}\right) \tag{3.1}$$

3.1.1. Loss Function

Backpropagation uses a loss or cost function for verification and learning, which means it is a supervised algorithm. This function indicates how the desired values deviate from the value found by the network. Categorical cross-entropy, CCE, is a commonly used loss function for multi-class classification problems. This loss is defined as

$$\mathcal{L}_{CCE}(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{3.2}$$

with y the true labels, $\hat{y}$ the predicted labels, and N the number of samples in y and $\hat{y}$.

3.1.2. Stochastic Gradient Descent

An optimizer is used to minimize this loss function in order to train the model. Stochastic gradient descent, SGD, is a simple optimizer, this is an iterative optimization algorithm which looks for local minima and changes the weights accordingly. The training of a model can be sped up with the use of batches, instead of updating the weights based on the whole data set, smaller parts of the input data are used to calculate intermediate weight updates. This speeds up the training process by providing more frequent weight updates for a set of data, however, this gradient is not as accurate as a full epoch gradient, but the training speed increment compensates for the slight inaccuracy. Before applying the change to the weights, the sum of the gradient matrix with respect to the weight matrix $\mathbf{W}_t$ of the mini-batch is divided by the batch size $|B|$. The learning rate η is a proportionality hyperparameter for the training speed.

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \frac{\eta}{|B|} \sum_{\forall s \in B} \nabla_{\mathbf{W_t}} \mathcal{L}(\mathbf{W}_t) \tag{3.3}$$

3.1.3. Momentum

A commonly used optimization technique is Momentum. This method provides a solution for the tendency of SGD to zigzag, especially with a small batch size. It replicates the concept of momentum from physics, in order to change the speed of a moving object, a certain force has to be applied for a certain amount of time. In the case of SGD if the weights are changing in one specific direction, it is more likely to keep moving in that direction. Providing the change of weight a direction of momentum makes the training process in relation to the loss function more analogous to a physical ball rolling smoothly into a valley rather than zigzagging towards the bottom. The weight update process with momentum is described by

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \mathbf{V}_{t+1} \tag{3.4}$$

where $\mathbf{V}$ is the velocity matrix, defined as

$$\mathbf{V}_{t+1} = \mu \cdot \mathbf{V}_t - \frac{\eta}{|B|} \sum_{\forall s \in B} \nabla_{\mathbf{W_t}} \mathcal{L}(\mathbf{W}_t) \tag{3.5}$$

3.1.4. Learning rate decay

Learning rate decay is another technique to adjust the learning rate during the training process. The learning rate determines how quickly the model parameters are updated in response to the error gradient calculated during backpropagation. The idea behind learning rate decay is to gradually reduce the learning rate over time as the training progresses [32]. This allows the model to make more significant updates in the beginning when the parameters are far from their optimal values, and smaller updates as it gets closer to convergence. By doing so, learning rate decay can help improve convergence, prevent overfitting, and lead to better model performance.

3.1.5. Forward and Backward Pass

Let the variables be defined as the following for the derivations and elaborations:
The loss function, denoted by $\mathcal{L}$, quantifies the discrepancy between predicted and target values. The learning rate of stochastic gradient descent (SGD) is represented by η and controls the step size during weight updates. The set B refers to a batch of samples, and $|B|$ represents the number of elements in that set and $s \in B$ each individual element of the set. The term (L) indicates the layer index of the neural network and is placed in the right corner. For example, the weight matrix $\mathbf{W}^{(L)}$ belongs to layer (L). The weight matrix, denoted as $\mathbf{W}$, contains the weights connecting the neurons in a neural network. Each entry, W_{ij}, represents a specific weight value.The bias vector, denoted as $\mathbf{b}$, contains the biases associated with the neurons in a neural network. Each entry, b_i, represents a specific bias value in this vector. The activation vector, represented by $\mathbf{a}$, consists of the activations of the neurons in a neural network. Each entry, a_i, corresponds to the activation of a particular neuron. This is typical an ReLU, softmax or sigmoid function:

$$ReLU(x) = \max(0, x), \quad softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad sigmoid(x) = \frac{1}{1+e^{-x}} \tag{3.6}$$

The linear combination vector $\mathbf{z}$ combines the input, weights, and biases in a neural network as shown in 3.7. Each vector component, z_i, represents an entry of this combination. The input vector of the neural network is denoted as $\mathbf{x}$, with each vector component x_i representing a specific input entry value. The output vector of the neural network is represented by $\mathbf{y}$, with each vector component entry y_i corresponding to an output value.
The first step of backprop is traversing the input data through the network like:

$$\mathbf{z}^{(L-1)} = \mathbf{W}^{(L-1)}\mathbf{a}^{(L-2)} + \mathbf{b}^{(L-1)} \tag{3.7}$$

$$\mathbf{a}^{(L-1)} = \sigma(\mathbf{z}^{(L-1)}) \tag{3.8}$$

With the first activation $\mathbf{a}^{(0)}$ as the input vector $\mathbf{x}$ and the last activation vector is $\mathbf{a}^{(L)}$ as the output vector $\mathbf{y}$ as shown in Fig. 3.1. This could be recursively forwarded until the last layer in the network:

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)} \tag{3.9}$$

$$\mathbf{a}^{(L)} = \sigma(\mathbf{z}^{(L)}) \tag{3.10}$$

Once the final output is obtained, the loss and gradient can be determined in relation to the weights. First, the gradients of the weights of the most nearby layer will be determined as follows,

$$\frac{\partial \mathcal{L}}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \tag{3.11}$$

This represents the summation of all chain rules between all possible paths from $\mathcal{L}$ toward $a_k^{(L)}$. Here n_L represents the number of neurons of layer L. The following relation is also recursive by nature but in the backwards direction, therefore backward pass with a shift in layer index form L to $L-1$:

$$\frac{\partial \mathcal{L}}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_{L-1}-1} \frac{\partial z_j^{(L-1)}}{\partial a_k^{(L-2)}} \frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}} \frac{\partial \mathcal{L}}{\partial a_j^{(L-1)}} \tag{3.12}$$

However, this backwards recursion can only continue when all the derivatives of the loss in relation to all the activation functions of that specified layer are calculated. In this case $\frac{\partial \mathcal{L}}{\partial a_k^{(L-1)}}$ for all possible index k in that layer L. This recursive relation continues until the first layer. In the end, the weights (or bias) can be obtained for each layer while the backward pass progresses:

$$\frac{\partial \mathcal{L}}{\partial W_{kn}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial W_{kn}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \tag{3.13}$$

Both passes could potentially be more rigorously verified by a proof by induction. Using this result, the local mini-batch stochastic gradient is described by:

$$W_{ij,t+1}^{(L)} = W_{ij,t}^{(L)} - \frac{\eta}{|B|} \sum_{\forall s \in B} \frac{\partial \mathcal{L}}{\partial W_{ij,t}^{(L)}} \tag{3.14}$$

Here, the weight matrix entry of layer number L, $W_{ij,t}^{(L)}$ is updated into $W_{ij,t+1}^{(L)}$ after the update iteration with t as the discrete recursive iteration variable. The concept of this update mechanism is discussed in section 3.1.2.

3.1.6. Problems With Backpropagation

Despite the success of backprop, the learning algorithm has some bottlenecks in application to resource-constrained hardware. One is memory usage, backprop requires storage for all the intermediate values, the input data, activations, and gradients for both the forward and backward pass. This can require a significant amount of memory, especially for neural networks with many hidden layers. Resource-constrained hardware, such as microcontrollers or embedded systems, often have limited memory capacities, making it difficult to accommodate the memory-intensive computations of backpropagation. Another one is computational complexity. Backprop involves computational intensive operations such as matrix multiplication, vector multiplication or element-wise operations. These operations require substantial computational power, including floating-point arithmetic and memory bandwidth. Resource-constrained hardware, such as low-power devices or edge devices, typically have limited processing capabilities and may not be optimized for these types of computations. Another problem is energy consumption, training neural networks using backpropagation can be computationally demanding, leading to high energy consumption. Resource-constrained hardware, particularly battery-powered devices, aim for energy efficiency to maximize battery life. The energy requirements of backpropagation algorithms may exceed the available energy budget, making them unsuitable for such hardware. Finally, backprop is not biologically plausible [16],[11]. The algorithm learns global information rather than local information. Furthermore, the back and forward pass does not resonate with how humans process information, neurons learn based on the information that is provided by their environment. This is a continuous process. Human perception does not stop or black out for a moment because the whole cortex system has to backwards pass the errors obtained by the processed information in order to learn.

3.2. TinyML

TinyML refers to low-level machine learning frameworks for deployment on low-cost hardware [25]. It enables the power of the machine learning world on smaller IoT devices. However, to implement those models, a significant reduction of resolution is necessary to implement them on those small low-level devices. This resolution reduction can be accomplished by quantization and pruning [25]. Quantization has proven to be provided to perform equivalent to full resolution models [29]. It can be subdivided into multiple methods. The first one is post-training quantization. Post-training quantization maps the floating point values onto a fixed point 8-bit value, which gives significant power and computational gains while potentially risking the neural network's performance [12]. This method can reduce the size by 4 times and speed up the inference process by 2 to 3 times. Another method is quantization-awareness training which simulated the quantization in the learning process by clamping, fusion and special layers. A relatively new method is banalized neural networks where the weight and activation are reduced to only one bit which provides about 8.5 to 19 speed-ups and 8 times memory reduction [25]. Another reduction method is pruning, pruning is one such method wherein unimportant parameters from a trained model are removed to reduce model size [5]. In the case of pruning the weights, it can improve the speed about 4 times and memory about 5 to 10 times. Those compression and dimensionality reduction methods could be utilized for better performance of the implementation.

4 Forward-Forward Algorithm

The forward-forward algorithm was proposed by Hinton [15] as an alternative to backpropagation. The main benefits of forward-forward over backpropagation are that it may be a better representation of biological learning and that it could be a suited algorithm for resource-constrained hardware.
Backpropagation has proven successful in deep learning [24], but its plausibility as a model for how the cortex learns is questionable. Moreover, it requires explicit propagation of error derivatives and the storage of neural activities for subsequent backward passes, which is not supported by evidence in the cortex. These requirements pose challenges for implementing backpropagation on low-cost hardware, which often has limited memory and computational resources.
The forward-forward algorithm offers a potential solution to these challenges. It replaces the forward and backward passes of backpropagation with two forward passes, eliminating the need for explicit error propagation and storage of neural activities. This feature makes the algorithm suitable for scenarios where low-cost hardware constraints limit the availability of resources for complex computations and memory operations. In the forward-forward algorithm, each layer of the neural network has its own objective function, aiming to maximize the goodness of positive, correct, data and minimize the goodness of negative, incorrect, data generated by the network itself or provided by supervision. By optimizing these objective functions through the forward passes, the algorithm adjusts the weights of the network to improve its performance on positive data and suppress the generation of negative data.
A function that could be suited has been proposed by [15] for layer-based goodness determination:

$$p(positive) = \sigma\left(\sum_j y_j^2 - \theta\right) \tag{4.1}$$

However, other functions can be used as a 'goodness' function or loss function.
Between the layers of the network, normalization is applied. This causes the second layer to only react to the relative activation of the output. The first layer thereby does not spoil a high or low total activation information for the second layer.
The forward-forward algorithm can be implemented in a supervised and unsupervised manner. The approach for supervised learning used by [15] is to embed the label into the input data itself. In the case of the MNIST dataset [6], this could be done by encoding the label into the upper left, see Fig. 4.1. The same logic is applied after training. During inference, the input is copied and marked with all the available labels in a one-hot encoded manner [16] and then fed into the network one by one. The input with the encoded label that causes the highest activation over all the layers summed then corresponds to the predicted output, see Fig. 4.2.

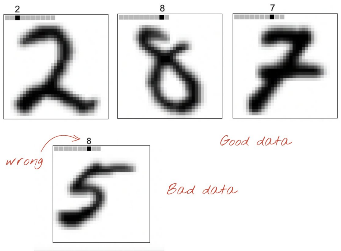


Figure 4.1: Label encoding good vs bad. [9]

4.1. Analysis Of Existing Implementations

Multiple routes for implementation could be considered. However, this document focuses mostly on the application of high-level frameworks of machine learning like TensorFlow. The framework has released a low-level framework, TensorFlow Lite, which is a TinyML framework. TensorFlow has proven to be a functional high-level module for neural network training. However, the degrees of freedom the module has in regard to low-level implementation for non-standard algorithms in TensorFlow Lite has proven to be not that flexible. Therefore, a list of implementations of the forward-forward algorithm have been studied. Those are all supervised versions. One is the custom model by Keras for high-level implementation [22]. However, this model proved to be unsuited for implementation in TensorFlow lite. Therefore, this custom object and layer-typed implementation were not suited for a microcontroller inference implementation. TensorFlow Lite does not seem to be quite compatible with customized models, layers and learning algorithms in general. Another implementation [1] which is based on a PyTorch implementation [21] of the same concept bypasses the tendency to train everything with backprop by pretending that every layer of the forward-forward model is a model itself. This is technically not backprop because it does not extend to multiple layers. It is single-layer-based gradient descent and not multi-layered, which is the case in backprop. The algorithm trains by first training the first layer, then feeding forward all the training data through the first layer and normalizes it to train the second layer and so on. This implementation avoids the use of TensorFlow models and only uses layers. These are indeed convertible to TensorFlow Lite, which is not the case for the other implementations. However, because every layer has its own TensorFlow Lite model, these models have to be pipelined one after another on the microcontroller. Consequently, normalization has to be implemented manually between the connection of the layers. This approach might take away some of the optimization benefits that are embedded in one multiple-layered model. This approach would result in a slight difference in implementation on the microcontroller between the backprop models and the forward forward models because backprop will be one model and forward multiple layer-based model placed one after the other manually.

4.2. Mathematical Derivation

In order to efficiently implement the supervised version of the forward-forward algorithm, it is important to comprehend the concept behind the algorithm. This is especially useful for the other subgroups. Therefore, a few high-level implementations have been studied for a more conceptual implementation independent of high-level frameworks. This could be more suitable for low-level implementations, such as on a MCU or FPGA. The studied implementations [1, 21, 22] thus far use Softplus ($f(x) = \ln(1+e^x)$) as a loss function with an input parameter ϕ which is the defined by the sum of all the squared activations averaged minus a constant θ:

$$\mathcal{L} = \ln(1+e^{\phi}), \quad \phi = \frac{1}{N}\sum_{j}^{N} y_j^2 - \theta \tag{4.2}$$

This is a deviation from the loss function proposed by Hinton [15, 17, 14], see equation 4.1.
This could be due to the inconvenient behaviour of the derivative of the sigmoid for large input values, which is not convenient in the case of forward forward due to the binary nature of the training mechanism.
Those implementations solved the problem of positive data learning and negative data learning by introducing two types of ϕ functions. One for the positive data and one for the negative data.

The ϕ_{pos} will be defined as $-\phi$ and ϕ_{neg} will be defined as ϕ because this would force positive data which is correctly labelled to iteratively converge to the positive side and visa-versa for the negative data that is incorrectly labelled. See Fig. 4.1 for a clear difference between supervised 'positive', 'good', correct data and 'negative', 'bad', incorrect data.

$$\phi_{pos} = -\frac{1}{N}\sum_{j}^{N} y_{j,pos}^2 + \theta, \quad \phi_{neg} = \frac{1}{N}\sum_{j}^{N} y_{j,neg}^2 - \theta \tag{4.3}$$

This gives for positive data (correctly labelled data) input $\mathbf{x}_{pos}$ that results in the layer-based output $\mathbf{y}_{pos}$, and for negative data input $\mathbf{x}_{neg}$ (incorrectly labelled data) that results in the layer based output $\mathbf{y}_{neg}$, resulting in a positive, and negative data combined loss where both are processed during training at the same time:

$$\mathcal{L} = \ln(1 + e^{\phi_{pos}}) + \ln(1 + e^{\phi_{neg}}) \tag{4.4}$$

However, theoretically, it could also be one after another.
Given a for the activation function, ψ for the linear combination of the weight inputs and biases:

$$y_i = a(\psi), \quad \psi = \sum_{j}(w_{ij}x_j + b_j), \tag{4.5}$$

This results in the following derivation for the gradient updates, where $\mathbf{x_{pos}}$ and $\mathbf{y_{pos}}$ are correctly labelled data and $\mathbf{x_{neg}}$ and $\mathbf{y_{neg}}$ are incorrectly labelled data:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial \phi}\cdot\frac{\partial \phi}{\partial y_i}\cdot\frac{\partial y_i}{\partial \psi}\cdot\frac{\partial \psi}{\partial w_{ij}} = \Delta W_{ij} \propto \sigma(\phi_{neg})\cdot a'(\psi_{neg})\cdot x_{j,neg}\cdot y_{i,neg} + \sigma(\phi_{pos})\cdot a'(\psi_{pos})\cdot x_{j,pos}\cdot y_{i,pos} \tag{4.6}$$

$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \phi}\cdot\frac{\partial \phi}{\partial y_i}\cdot\frac{\partial y_i}{\partial \psi}\cdot\frac{\partial \psi}{\partial b_i} = \Delta b_i \propto \sigma(\phi_{neg})\cdot a'(\psi_{neg})\cdot y_{i,neg} + \sigma(\phi_{pos})\cdot a'(\psi_{pos})\cdot y_{i,pos} \tag{4.7}$$

The result clearly has some Hebbian characteristics due to the clear dependency on the in and outputs. However, a is commonly a ReLU function. The derivative of the ReLU is a step function, resulting in the following simplification because y_i is a ReLU function that shadows the affect of a' step function.

$$\Delta W_{ij} \propto \sigma(\phi_{neg})\cdot x_{j,neg}\cdot y_{i,neg} + \sigma(\phi_{pos})\cdot x_{j,pos}\cdot y_{i,pos}, \quad \Delta b_j \propto \sigma(\phi_{neg})\cdot y_{i,neg} + \sigma(\phi_{pos})\cdot y_{i,pos} \tag{4.8}$$

Where ΔW_{ij} and Δb_j are entries of the differences between in the weight matrix and bias vector for each iteration. This generalizes to matrix and vector form:

$$\Delta\mathbf{W} \propto \sigma(\phi_{\text{neg}})\mathbf{y}_{neg}\mathbf{x}_{neg}^T + \sigma(\phi_{\text{pos}})\mathbf{y}_{pos}\mathbf{x}_{pos}^T, \quad \Delta\mathbf{b} \propto \sigma(\phi_{\text{neg}})\mathbf{y}_{neg} + \sigma(\phi_{\text{pos}})\mathbf{y}_{pos} \tag{4.9}$$

This result could be substituted into a SGD algorithm. Clearly, a Vanilla Hebbian element is embedded in this theoretical derivation of the forward-forward algorithm. It seems to have Hebbian characteristics during training when positive data is provided and anti-Hebbian when negative data is provided. However, due to ϕ in the sigmoid function, it is not fully local but layer local as visualised in Fig. 4.2.

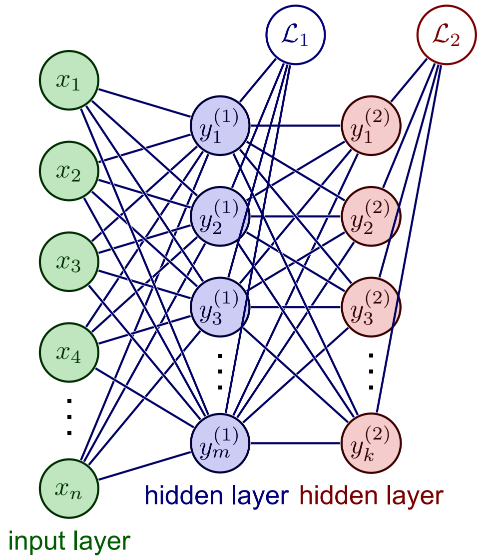


Figure 4.2: Forward-Forward network with layer-based loss function. Adapted from [23].

5 Hebbian Learning

Hebbian learning is a learning rule in neuroscience and artificial neural networks that describes how synaptic connections between neurons are strengthened or weakened based on their activity patterns [13]. The fundamental idea behind Hebbian learning is often summarized as ”fire together, wire together” [28]. According to Hebb’s postulate, when a presynaptic neuron repeatedly and persistently stimulates a postsynaptic neuron, the connection between them is strengthened. This strengthening is believed to be the basis for learning and memory formation in the brain [27]. Hebbian learning is typically described using a simple mathematical rule known as the Hebbian learning rule. The rule states that if the presynaptic neuron consistently activates the postsynaptic neuron, then the strength of the synaptic connection between them should be increased. Conversely, if the presynaptic neuron is consistently inactive while the postsynaptic neuron fires, the synaptic connection should be weakened. In artificial neural networks, Hebbian learning is often implemented as a form of unsupervised learning, where the network adjusts its weights based solely on the correlation between input patterns and their corresponding outputs. This type of learning is useful for tasks such as pattern recognition, clustering, and self-organization. It is important to note that Hebbian learning is a simplified model of synaptic plasticity, and there are other factors and mechanisms involved in real neural networks such as Spike-timing-dependent-plasticity. However, this simplicity and the idea of learning by synaptic locality make the Hebbian learning rules a good contender for a more biologically plausible and more local alternative to backpropagation, especially in relation to resource-constrained hardware. Furthermore, there are some indications that Hebbian learning in some specific cases could outperform backprop [11]. This makes it an interesting algorithm for further investigation.

5.1. Spike Time Dependent Plasticity

Spike Time Dependent Plasticity (STDP) states that there is a timing relation between the pre and post-synaptic neurons which causes the strengthening or weakening of the connection. This neurologically-based approach could be modelled. General modelling of spike time-dependent neural networks is done with spike trains [7]. Or a sum of Dirac delta functions for different time instances [7]. Presynaptic neurons fire those time-dependent trains into the postsynaptic neuron. The postsynaptic neuron integrates those over those values until a certain threshold is reached. Then the postsynaptic neuron fires a spike itself. If this spike is slightly later than a respective presynaptic neuron, the weight increases and vice versa when the postsynaptic neuron is slightly later than the postsynaptic neuron. This behaviour of the pre and postsynaptic neuron is shown in Fig. 5.1.

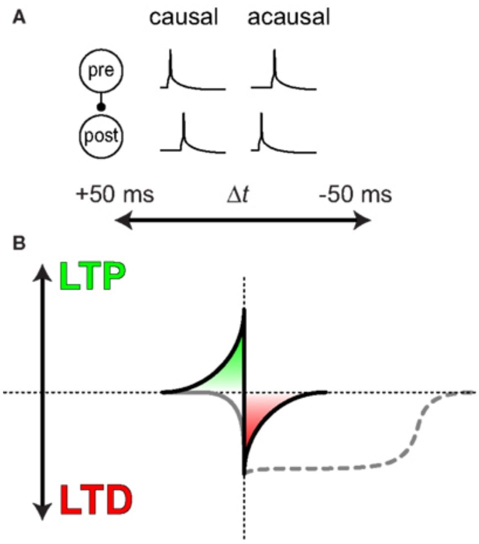


Figure 5.1: Behaviour STDP. [18]

5.2. Vanilla Hebbian

The tradition for of Hebbian learning is often expressed as $\Delta W_{ij} = \eta \cdot x_i \cdot y_j$ where x_i is the presynaptic neuron, y_j is the postsynaptic neuron and ΔW_{ij} weight update. It is not dependent on activation time differences between the neurons, like in STDP. The model implies that the connection is strengthened when both fire and reduced when they do not. However, one of the drawbacks of this learning method is the unboundedness of the value of the weights [11]. It could theoretically grow indefinitely, and this could cause problems in learning. Especially in resource-constrained hardware where the range of numeric representations within the supported data types is relatively small. Therefore, it is important for low-cost hardware to get some control over those potential weight ranges.

5.3. Oja's Learning rule

Oja's rule tackles this problem:

$$\Delta W_{ij} = \eta \cdot y_i \cdot (x_j - \alpha \cdot W_{ij} y_i)$$

Here η and α are training hyperparameters. In regular circumstances, Oja's rule aims to asymptotically normalize these synaptic weights ($\sum_j W_{ij}^2 = 1$) [28] and this normalization is done locally on neuron-based level [4] which is more biologically plausible than the layer local nature of forward-forward which has been discussed in Chapter 4. This makes Oja's rule far more interesting for the application on low-cost hardware than traditional vanilla.

5.4. Grossberg's Instar Learning Rule

Another rule is Grossberg's Instar Learning Rule which also prevents the problems of unbounded weights. The rule is described as follows:

$$\Delta W_{ij} = \eta \cdot y_i \cdot (x_j - \alpha \cdot W_{ij})$$

The rule has the same training hyperparameters as Oja's rule.

5.5. Sidenotes About Hebbian Learning

Due to its locality and simplicity, this learning method should be more suited for unsupervised learning. However, it is hard to obtain some useful information about the input data in this manner because it is unsupervised. One suggestion to make a Hebbian network supervised could be by adding a last supervised layer on the network at the end, which maps the learned patterns and features to corresponding correct labels. This last layer could be a one-layered backprop-trained network or a supervised forward-forward one.

6 Methodology

This chapter describes the methodology used to collect the results. Only backpropagation and forward-forward were tested using this methodology. Hebbian learning was not investigated, mainly due to time constraints.
Preliminary testing was performed on Hebbian learning in the same way as for backpropagation and forward-forward, however this preliminary testing showed that implementing and investigating Hebbian learning would be more challenging than forward-forward and likely would not perform as well as backpropagation and forward-forward. The main issues found during preliminary testing were that the available Hebbian implementations were very complex and often not in the form of a simple fully connected neural network and thus would not be comparable to a simple backpropagation network or a forward-forward network. Another consideration was the unexpected behaviour of the forward-forward algorithm. After these tests it was decided to spend more time on investigating forward-forward instead of trying to implement Hebbian learning. The results of the preliminary tests have been included in Chapter 7.

6.1. Environment

The models have been made in Python using TensorFlow [19] and were converted via TensorFlow Lite. TensorFlow is a machine learning framework, which allows the use of Keras [3]. TensorFlow and Keras simplify the development of models. Although, the main reason TensorFlow is chosen over other machine learning frameworks is TensorFlow Lite. TensorFlow Lite converts standard TensorFlow models to a format which can be used to run inference on edge devices, other frameworks do not offer similar functionality. TensorFlow Lite does not support on-edge training, this still has to be done manually.

6.2. Fashion MNIST Dataset

The dataset used to train the models is the Fashion MNIST dataset [30]. This dataset has similar characteristics to the MNIST dataset with handwritten [6], but is more challenging. These two datasets offer a very large train and test set, and due to their popularity they also function as a way to benchmark the results. A description of the classes of the dataset is provided in Table 6.1. For the models trained on the Fashion MNIST dataset, a requirement of at least 80% was set. The accuracy requirement is based on the published accuracy benchmark given in [30]. From the 129 entries, 83 are above 80%, while only 37 out of 129 are above 85%. An accuracy of 80% is thus about average, although a result of 99.7% was claimed by [26].

6.3. Metrics

The metrics that were used to compare the performances of the models to each other is shown in Table 6.3. Minimum amount of epochs required was used to limit the training time of the models for both Google Colab and on hardware. 200 epochs was chosen as an arbitrary cut off, where it is expected that the improvement from extra epochs has plateaued. From the plotted loss it is then determined how

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Table 6.1: A description of the Fashion MNIST classes. [31].

many epochs are required to get within 2% of the minimum validation loss. From preliminary testing it was found that forward-forward trained models train significantly slower than backpropagation trained models, for this reason the amount of epochs is reduced to 100. 100 epochs with forward-forward still trains slower than 200 epochs of backpropagation, but it was also found during preliminary testing that forward-forward requires more epochs to train. It is also mentioned by [15] that forward-forward takes more epochs to reach a similar performance to backpropagation. The loss was used since the method used to train the forward-forward models, `train_on_batch()`, only returns the loss. The `model.fit()` function is able to return both the loss and the accuracy.

The results gathered from the models have been collected in Google Colab[1] with automated notebooks. Google Colab was used to provide a test environment with fixed hardware for different users. Google Colab does not allow the user to choose the hardware, but consistently provides a virtual environment with comparable specifications, the different hardware configurations that were encountered are listed in Table 6.2. The most common configuration is the one listed in the first column, the other configurations were very rarely encountered. The hardware configuration was collected via either `!cat /proc/cpuinfo` or `!lscpu`. The metrics were collected under the assumption that all hardware configurations have equal performance. Since Colab does not give the specific CPU model, only `Intel(R) Xeon(R) CPU`, it was not possible to identify performance differences. The L3 cache size and clock speed did show differences, however it cannot be assumed that performance differences are not cancelled out by a more powerful CPU core.

A fixed hardware setup is necessary to accurately compare the training and inference time between the models. After testing the standard TensorFlow models and the TensorFlow Lite models, the Lite models were verified by running inference on a microcontroller, inference on the FPGA was done manually with the weight and biases of the TensorFlow Lite model.

It is also possible to connect to a Colab environment with a NVIDIA Tesla T4 GPU, but these were often unavailable, however TensorFlow did not make use of the GPU even when available, so the notebooks were all run without a GPU.

CPU model	Intel(R) Xeon(R) CPU	Intel(R) Xeon(R) CPU	Intel(R) Xeon(R) CPU
CPU frequency	2.20 MHz	2.30 MHz	2.00 MHz
CPU cores	1 core, 2 threads		
L3 cache size	55 MB	45 MB	38.5 MB

Table 6.2: Google Colab hardware specifications.

6.4. Hardware Limitations

The hardware used to verify the models are a Teensy 4.1 microcontroller and a Digilent ZedBoard with a Xilinx Zynq-7000 System-on-Chip. Google colab does not impose any hardware limitations for developing and testing the models. The Teensy and the ZedBoard do have some hardware limitations that have to be accounted for. The Teensy imposes a memory limit, running inference of pre-trained

[1] https://colab.research.google.com/

Metrics	Description
Accuracy	The accuracy of the model on the test dataset
Training time	The time to train the complete model
Inference Time	The average time to predict one sample
Memory required	The model size of the converted TensorFlow Lite model, split into data buffer and non-data buffer
Epochs required	An estimate, based on the loss, of how many epochs are required to get within approximately 2% of the minimum loss of the same model trained for 200 epochs for backpropagation or 100 epochs for forward-forward with a limit for the training time of 10 minutes.
Accuracy after minimum amount of epochs required	The accuracy of the model after training for the minimum amount of epochs required

Table 6.3: The metrics used to compare the models.

```
--------------------------------------------------------------
          Model size:      27568 bytes
    Non-data buffer size:       1888 bytes (06.85 %)
  Total data buffer size:      25680 bytes (93.15 %)
    (Zero value buffers):          0 bytes (00.00 %)

* Buffers of TFLite model are mostly used for constant tensors.
  And zero value buffers are buffers filled with zeros.
  Non-data buffers area are used to store operators, subgraphs and etc.
  You can find more details from https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema.fbs
```

(a) The reported size of the TensorFlow Lite model for a single-layer network with 32 neurons.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 32)                25120

 dense_3 (Dense)             (None, 10)                330

=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
```

(b) The model summary of a single-layer network with 32 neurons.

Figure 6.1: TensorFlow Lite model size (a) and the model summary (b) as reported by TensorFlow

models on these requires the model to be loaded into memory. The ZedBoard has 512 MB of DDR3 memory, while the Teensy only has 500 KiB of memory, which can only partially be used to load in the model, the rest of the memory is taken up by the files needed to inference the model. The ZedBoard does not have this memory limitation, but it is limited by its use of fixed point arithmetic, this limits its capability in intensive multiplication operations.

6.4.1. Estimating the Model Size

When converting a TensorFlow model to a TensorFlow Lite model, the size of the resulting model is also reported, shown in Fig. 6.1a, the size is made up of a data buffer and a non-data buffer. The data buffer stores the weights and biases of the model. Fig. 6.1b shows the model summary given by TensorFlow, the number of parameters in total, Total params, and per layer, Param #, represent the amount of weights and biases of the model. Both of these are from the same single-layer network with 32 neurons. The total number of parameters can be found as

$$Total\,params = \sum_{i=1}^{n} L_{i,in} \times L_{i,out} + L_{i,out} \tag{6.1}$$

Layers	Neurons per layer (BP)	Neurons per layer (FF)
1	430	410
2	309	297
3	259	250

Table 6.4: The maximum hidden layer configuration that can fit in the microcontroller memory for backpropagation (BP) and forward-forward (FF). The amount of neurons per layer is the same for each layer.

where n is the total number of layers, counting the hidden layers and the output layer, and L_{in} and L_{out} are the input and output shape of layer i respectively. If the same amount of neurons is used for each layer, this simplifies to 6.2 for Fashion MNIST, with n the amount of hidden layers and L the neurons per layer.

$$Total\,params = (784 + n + 10) \times L + (n-1) \times L^2 + 10 \tag{6.2}$$

Forward-forward does not require an output layer, this then simplifies to 6.3

$$Total\,params = (784 + n) \times L + (n-1) \times L^2 \tag{6.3}$$

For estimating the neurons per layer for the large models, a 5% margin was used. After loading the required files into the Teensy memory, backpropagation leaves $351KiB$ of memory available, including the 5% margin this leaves $334KiB$ for the model. For forward-forward this is $331KiB$ and $316KiB$ with a 5% margin. The neurons per layer were then found by equating 6.2 or 6.3 to the available memory and solving for L. Table 6.4 lists these values for the neurons per layer for a single-, two-, and three-layered network.

6.5. Developing the Models

The models have been developed using existing implementations of the respective algorithms. For backpropagation, the Fashion MNIST model from the TensorFlow Keras tutorial for classification was used[2]. The forward-forward models were generated based on a PyTorch implementation by Pezeshki[3] and a TensorFlow implementation by Rajabi[4]. For Hebbian learning the implementation by Miconi [20] was used, this was also used by Gupta [11].

First some preliminary tests were done with these implementations, this consisted of first reproducing the claimed results, after which some simple models were made, these provided more insight into training time and general performance to be expected, as a last step, these simple models were attempted to be converted to a TensorFlow Lite model using default optimization, this gives a quantized model with 8-bit weights. The implementations that successfully passed these preliminary tests were then modified to include automated testing and to be more versatile in training different models. These were then used to test models in TensorFlow and TensorFlow Lite and to inference on the Teensy microcontroller and the ZedBoard FPGA.

[2] https://www.tensorflow.org/tutorials/keras/classification

[3] https://github.com/mohammadpz/pytorch_forward_forward

[4] https://github.com/amirrezarajabi/Tensorflow-Forward-Forward

7 Results

This chapter contains all the metrics collected from the developed models. Each model is tested in TensorFlow (TF) and TensorFlow Lite (TF Lite). All models except the forward-forward reference model, due to model size limitations, have been inferenced on the Teensy microcontroller (MCU). Some backpropagation models have also been inferenced on the Digilent ZedBoard (FPGA). Forward-forward models could not be inferenced on the FPGA due to it requiring normalization of the output of the layers, this was too computationally expensive for the FPGA.

7.1. Backpropagation

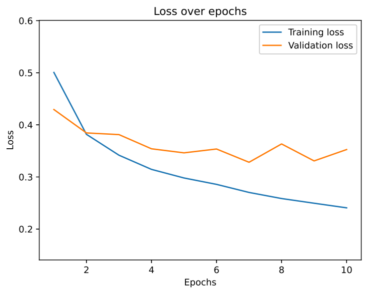
All results shown in this section have been provided with the notebook in Appendix B. The backpropagation trained models consist of an input layer with a size of 784, a flattened Fashion MNIST sample, and an output layer of 10 neurons, one neuron for each class of the dataset. The hidden layers refer to the layers in between the input layer and the output layer. The models were converted into a model with 8-bit integer input and outputs.

7.1.1. Reference Model

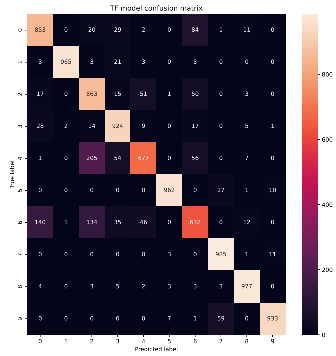
The reference model for backpropagation is a model with a single hidden layer with 128 neurons and the default Adam optimizer, trained for 10 epochs. Table 7.1 shows the results of this model in TF, TF Lite, on the MCU. Fig. 7.1a shows the loss during training over the epochs. Fig. 7.1b shows the confusion matrix of the TF model. This model does not predict class 4 and class 6 well, it has a recall for these classes of 68% and 63% respectively. This is shown in Fig. 7.1b where class 4 is predicted incorrectly as class 2 for 205 out of 1000 samples. Class 6 is mostly misclassified as classes 0 or 2. Fig. 7.1c shows the confusion matrix for the converted TF Lite model, a similar pattern can be observed here. Fig. 7.1d shows the differences in predictions between the TF model and the TF Lite model, this difference is due to the quantization when converting from a TF model to a TF Lite model.

	TF	TF Lite	MCU
Epochs trained	10	-	
Accuracy (%)	87.71	87.66	87.66
Training time (s)	91.61	-	
Inference time per sample (μs)	121.59	46.45	248.25
Total model size (KiB)	-	101.73	239.90
Data buffer (KiB)	-	99.89	134.16
Non-data buffer (KiB)	-	1.84	105.74

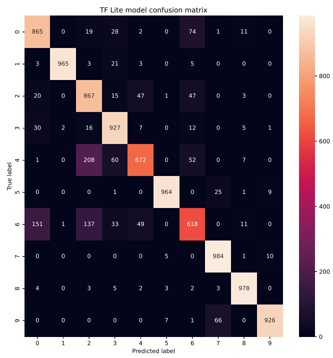
Table 7.1: The results of the reference model.



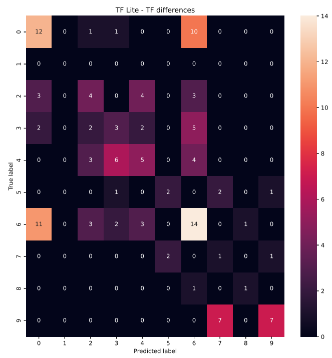
(a) The training and validation loss plotted over the epochs.



(b) The confusion matrix of the TensorFlow model.



(c) The confusion matrix of the TensorFlow Lite model.



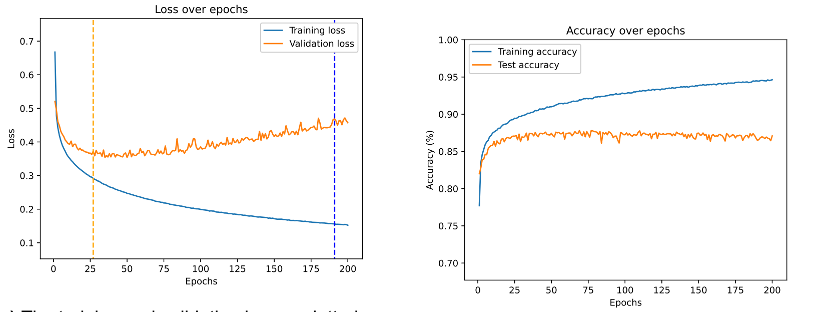
(d) The differences between the predictions of the TensorFlow and the TensorFlow Lite model.

Figure 7.1: The loss over the epochs and the confusion matrices for the reference model, a single layer model with 128 neurons in the hidden layer, trained for 10 epochs with backpropagation.

7.1.2. Small Single-Layer Model

For the small single-layer model, a size of 32 neurons for the hidden layer was chosen. This size was chosen in relation to the two-layer model discussed in the next section. The hyperparameters used for the other backpropagation models were changed from the reference model. Stochastic gradient descent was used as the optimizer instead of Adam, this allows easier implementation for on-device training. For the learning rate, the default value for stochastic gradient descent in TensorFlow, of 0.01, was used. Other hyperparameters for the optimizer are momentum and learning rate decay, the default optimizer does not use either of these. These are investigated in Section 7.1.4, 7.1.5, and 7.1.6. The batch size was set to 16, this was empirically found to be the optimal learning rate for on the MCU. A small batch was chosen as it allows faster learning, this is especially important on hardware with limited computing power.

This model was first run for 200 epochs, the corresponding loss and accuracy is plotted in Fig. 7.2a and 7.2b, these plots show that the loss and accuracy both peak between 30 and 40 epochs, after which the model starts overfitting. The training performance keeps rising, but the test performance starts to decrease. The amount of epochs required to achieve the minimum loss with a 2% margin is 30 epochs. The loss for the model trained for 30 epochs is shown in Fig. 7.3. Table 7.2 shows the metrics collected for this model. This model was also used to convert to TF Lite for inference on the MCU and FPGA, these results are included in Table. 7.2. Similar to the reference model, the accuracy of the TF Lite model is slightly lower than the TF model due to quantization. The TF Lite model has the same accuracy in Colab as on the MCU, this is expected, as they use the same TF Lite model for inference. The accuracy on the FPGA is lower than the TF Lite accuracy, this is because the FPGA does not use the TF Lite model for inference, instead it manually computes the output of the model via the weights and biases. The inference time of the TF Lite model is lower than that of the TF model, this shows the reduction in complexity of the model after quantization by TensorFlow. The MCU is less powerful than a computer, so as expected the inference time on the MCU is higher than the TF Lite model inferenced in Colab. The size of the model listed is reported by PlatformIO, this is a development tool provided with the microcontroller. The size reported by PlatformIO is significantly bigger than the TF Lite model size, this is because it includes all the files necessary to run the TF Lite model on the MCU.



(a) The training and validation losses plotted over 200 epochs, the vertical lines indicate the required epochs to train this model.

(b) The accuracy on the training set and test set plotted over 200 epochs.

Figure 7.2: The loss over the epochs for a single-layer model trained for 200 epochs (a) and the corresponding accuracy (b).

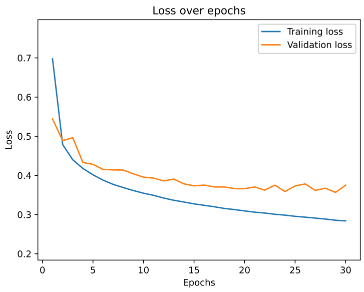


Figure 7.3: The loss of the final model, trained for 30 epochs

	TF	TF Lite	MCU	FPGA
Epochs trained	30	-		
Accuracy (%)	86.41	86.33	86.33	85.78
Training time (s)	142.2	-		
Inference time per sample (μs)	65.02	34.62	70.19	
Total model size (KiB)	-	26.42	165.90	
Data buffer (KiB)	-	25.08	60.16	
Non-data buffer (KiB)	-	1.34	105.74	

Table 7.2: The results of the single hidden layer model trained for 30 epochs.

7.1.3. Two-layer Model

The largest two-layer model that could be trained on the microcontroller is a model with two layers of 32 neurons each. This model was trained with the same hyperparameters as the single-layer model. The loss of the model over 200 epochs is plotted in Fig. 7.4, the amount of epochs required for this model was 22 epochs indicated by the orange dotted line. This model required less epochs to train than the single-layer model, even though it has more weights to train. The single-layer model is likely more restrictive in training, due to the low amount of neurons the optimizer can utilize. The results of this model are listed in Table 7.3. The results display a similar behaviour as for the single-layer network, the effect of TF Lite quantization is small and the TF Lite inference in Colab is equal to the inference on the MCU. The performance of the model has increased by about 1% for TF and TF Lite on both platforms, while the size of the TF Lite file has increased 2.85 KiB.

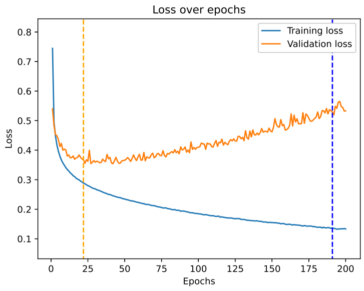


Figure 7.4: The loss of the two-layered model over 200 epochs, the dotted lines indicate the number of epochs where the 2% threshold is reached.

	TF	TF Lite	MCU	FPGA
Epochs trained	22	-		
Accuracy (%)	87.46	87.36	87.36	86.00
Training time	123.9	-		
Inference time per sample (μs)	72.34	37.47	75.2	
Total model size (KiB)	-	28.53	166.90	
Data buffer (KiB)	-	26.20	61.16	
Non-data buffer (KiB)	-	2.33	105.74	

Table 7.3: The results of two-layer model trained for 22 epochs.

7.1.4. Single-Layer with Momentum

The single-layer model in Section 7.1.2 did not utilize momentum and learning rate decay, however these can be used to improve the training time. These two hyperparameters will be investigated in the following three sections, starting with momentum, followed by learning rate decay, concluding with a combination of learning rate decay and momentum. Fig. 7.5 shows the loss over 200 epochs for the model trained with momentum. The epochs required to train this model is 12 epochs, this is significantly lower than the base single-layer model, however it also overfits the training data at a higher rate. The results for the model trained with 12 epochs is shown in Table 7.4. The accuracy of the trained model is very similar to the base single-layer model, but the training time has been reduced by 41.7%. Another notable result for this model is that the TF accuracy is 0.02% lower than the TF Lite model inferenced in Colab, while the TF Lite model inferenced on the MCU has the same accuracy as the TF model.

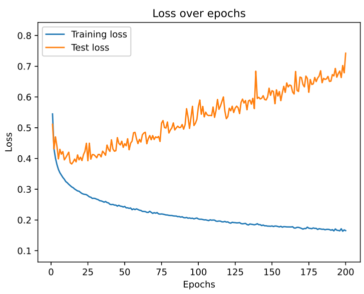


Figure 7.5: The loss over 200 epochs for a single-layer model with momentum.

	TF	TF Lite	MCU
Epochs trained	12	-	
Accuracy (%)	86.43	86.45	86.43
Training time	82.88	-	
Inference time per sample (μs)	69.10	24.83	70.35
Total model size (KiB)	-	26.42	165.90
Data buffer (KiB)	-	25.08	60.16
Non-data buffer (KiB)	-	1.34	105.74

Table 7.4: The results of the single-layer model trained for 12 epochs with momentum.

7.1.5. Single-Layer with Learning Rate Decay

Similar to the model trained with momentum, this model uses the base model discussed in Section 7.1.2. Now a learning rate decay is added, starting from a learning rate of 0.1 decaying every 200 batches by 5%, until a minimum learning rate of 0.01. The loss of this model is plotted in Fig. 7.6. The model requires 26 epochs to train. The results after 26 epochs are listed in Table 7.5. Looking at the results, it performs on par with the other base single-layer model and the single-layer model with momentum, however, its training time is significantly higher at 262.6 seconds, even though the amount of epochs is lower. This could be attributed to the additional task of keeping track of the learning rate and the number of batches.

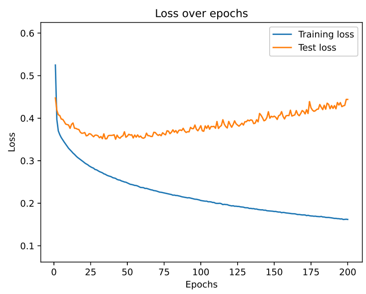


Figure 7.6: The loss of a single-layer model with learning rate decay.

	TF	TF Lite	MCU
Epochs trained	26	-	
Accuracy (%)	86.74	86.68	86.68
Training time	262.6	-	
Inference time per sample (μs)	86.97	30.55	70.10
Total model size (KiB)	-	26.42	165.90
Data buffer (KiB)	-	25.08	60.16
Non-data buffer (KiB)	-	1.34	105.74

Table 7.5: The results of the single-layer model trained for 26 epochs with learning rate decay.

7.1.6. Single-Layer with Learning Rate Decay and Momentum

This model combines the learning rate decay of Section 7.1.5 and the momentum of Section 7.1.4. The loss is plotted in Fig. 7.7 and the results are given in Table 7.6. The plotted loss shows a very different behaviour compared to the previously trained models, it drops very fast and almost immediately plateaus, however according to the epochs required metric defined in Chapter 6, this model requires 76 epochs to train, resulting in a training time over the limit of 10 minutes. Furthermore, the accuracy of this model, even after 76 epochs, is significantly lower at 80.51%. It is then clear that this combination of learning rate decay and momentum does not work. Another observation about this model is that, similar to the model from Section 7.1.4, the TF Lite accuracy is higher than the TF accuracy and the MCU inference result is again 0.02% lower than the TF Lite inference in Colab.

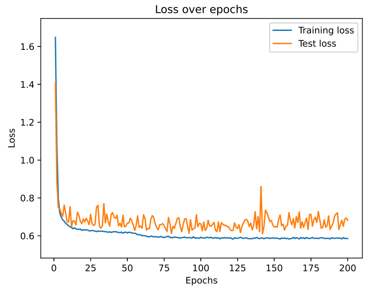


Figure 7.7: The loss for the single-layer model with learning rate decay and momentum plotted over 200 epochs.

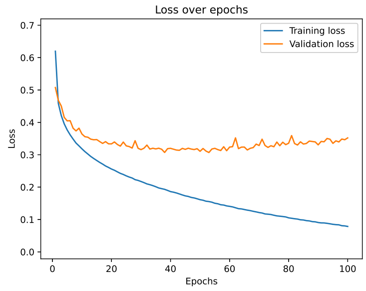


Figure 7.8: The loss plotted over 100 epochs for a single-layer model with 430 neurons in the hidden layer.

	TF	TF Lite	MCU
Epochs trained	76	-	
Accuracy (%)	80.51	80.60	80.58
Training time	682.96	-	
Inference time per sample (μs)	74.21	38.99	69.99
Total model size (KiB)	-	26.42	165.90
Data buffer (KiB)	-	25.08	60.16
Non-data buffer (KiB)	-	1.34	105.74

Table 7.6: The results of the single-layer model trained for 76 epochs with learning rate decay and momentum.

7.1.7. Largest Single-Layer Model

The largest pre-trained single-layer backpropagation model that was estimated to fit into the available memory of the MCU consists of one layer of 430 neurons. This model was trained with the same hyperparameters as the single-layer model from Section 7.1.2. Fig. 7.8 shows the loss during training again, this model was only trained until 100 epochs to save time, however it still shows that the test loss goes up after 60 epochs. This model required 37 epochs to train, the results are given in Table 7.7. The size on the MCU came out to be $475.9KiB$, $24.1KiB$ short of the $500KiB$ memory on the MCU.

	TF	TF Lite	MCU
Epochs trained	37	-	
Accuracy (%)	87.6	87.72	87.73
Training time	530.81	-	
Inference time per sample (μs)	137.95	49.48	819.45
Total model size (KiB)	-	337.09	475.90
Data buffer (KiB)	-	335.24	370.16
Non-data buffer (KiB)	-	1.85	105.74

Table 7.7: The results of the largest single-layer model that fits on the MCU.

7.1.8. Largest Two-Layer Model

The largest two layer model that can fit into the MCU memory was estimated to be two layers of 309 neurons each. This model trained faster than the Large single-layer model, requiring only 20 epochs to train. The results of this model are shown in Table 7.8.

	TF	TF Lite	MCU
Epochs trained	20	-	
Accuracy (%)	88.13	88.18	87.73
Training time	322.11	-	
Inference time per sample (μs)	143.11	50.83	819.45
Total model size (KiB)	-	337.72	475.90
Data buffer (KiB)	-	335.39	370.16
Non-data buffer (KiB)	-	2.33	105.74

Table 7.8: The results of the largest two-layer model that fits on the MCU.

7.1.9. Largest Three-Layer Model

The largest three layer configuration that fits in the MCU memory was estimated at three layers of 259 neurons each. The results for this model are show n in Table 7.9.

	TF	TF Lite	MCU
Epochs trained	16	-	
Accuracy (%)	87.90	87.99	87.97
Training time	326.90	-	
Inference time per sample (μs)	118.06	57.64	1010.86
Total model size (KiB)	-	337.87	475.90
Data buffer (KiB)	-	335.02	370.16
Non-data buffer (KiB)	-	2.85	105.74

Table 7.9: The results of the largest three-layer model that fits on the MCU.

7.1.10. Summarizing the Backpropagation Results

The overall behaviour of the backpropagation models was very similar. The models did not require a long training time and started overfitting when training for a high amount of epochs. The conversion from TF to TF Lite consistently resulted in TF Lite models with a similar accuracy compared to the original TF model, however for the small models the conversion resulted in a slight drop in accuracy, while for the large models this resulted in a slight increase. The TF Lite models inferenced on the MCU either had the same accuracy as the TF Lite file inference in Colab or were off by 0.02 percentage points. This deviation is very likely the result of rounding errors on the MCU, as it only reports the accuracy up to 2 decimals. The backpropagation model with the highest accuracy is the largest two-layer model from Section 7.1.8, however the three-layer model is almost identical in performance and training time. The performance of the large single-layer model is not far behind, but this model requires at least a 60% longer training time.

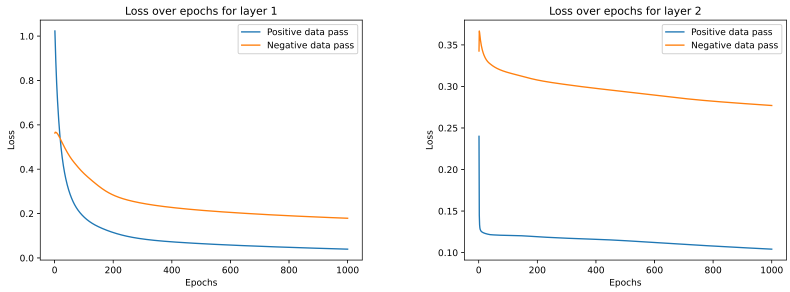
7.2. Forward-Forward

The results for the forward-forward algorithm follow a similar structure as the previous section. First a reference model is provided as a baseline, then small models are presented that can be trained on the MCU, finally, results of models are presented that maximize the memory of the MCU. In addition, some experiments with different hyperparameters per layer have been done. The TF Lite models for forward-forward have 32-bit float inputs and outputs, the inputs and outputs could not be converted to 8-bit integers.

7.2.1. Reference Models

For the forward-forward algorithm there are two reference models, the first reference model is the model provided with the TensorFlow implementation of the forward-forward algorithm by [1]. This model

consists of two layers of 500 neurons each. This model is trained with the default Adam optimizer, a batch size of 1000, and 1000 epochs. The losses during training are shown in Fig. 7.9, since the layers are trained independently, the losses are also independent per layer. The results for this model are summarized in Table 7.10. The table also lists the accuracy of only the output of the first layer, since the classification of forward-forward is based on the output of each layer it is possible to investigate each layer separately.
For inference each input is encoded once with each label, the classification is then equal to the encoded label that resulted in the highest accumulated goodness of all layers. Hinton [15] mentions that the output of the first hidden layer should not be used for this, however this makes it impossible to test single-layer models. For this reason the inference is kept as is in [1], where each layer contributes to the accumulated goodness.



(a) The positive and negative loss of the first layer of the model. **(b)** The positive and negative loss of the second layer of the model.

Figure 7.9: The positive and negative losses over the epochs for both layers of the reference model.

	TF	TF first layer only	TF Lite	TF Lite first layer only
Epochs trained	1000	-		
Accuracy (%)	87.43	87.67	87.36	88.07
Training time	2435.13	-		
Inference time per sample (μs)	2430.55	1416.96		
Total model size (KiB)	-		633.95	386.32
Data buffer (KiB)	-		631.06	384.87
Non-data buffer (KiB)	-		2.89	1.45

Table 7.10: The results of the forward-forward reference model.

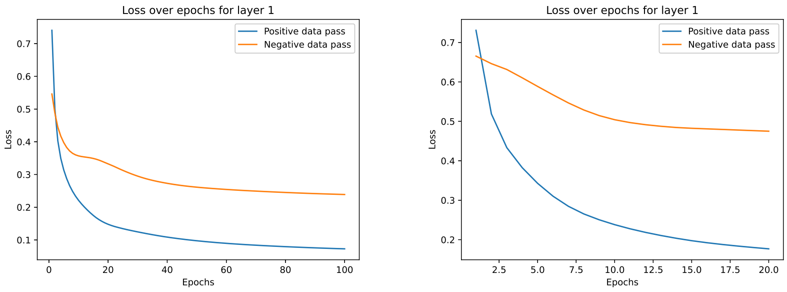
The second reference model is the model described in [15], this is a network with four layers of 2000 neurons each. This model is trained for 60 epochs with a batch size of 500 and multiple learning rates for different parts of the training. This model was first attempted to be trained using the same hyperparameters as the first reference model with the TensorFlow implementation. However when attempting to inference this model in Colab after training, the notebook crashes due to the inference process exceeding the available ram in Colab.

7.2.2. Hyperparameters

For the forward-forward algorithm a few hyperparameters were adjusted, the batch size was increased to 32 and the learning rate was increased to 0.1. From preliminary testing with the Teensy it was found that with the hyperparameters used for backpropagation, the model cannot be trained on-edge. The amount of epochs the models are trained for was also adjusted for forward-forward. Due to the need for positive and negative data passes in combination with the small batch size, the training time has significantly increased compared to the same model configurations in backpropagation. All forward-forward models were trained with the amount of epochs estimated to take 10 minutes to train.

7.2.3. Small Single-Layer Network

The first model that was trained was a small single layer network with 32 neurons, similar to the model discussed in Section 7.1.2. The losses for the small single-layer model are plotted in Fig. 7.10, these plots show that training up to 80 epochs would better minimize the losses, however this would take 36 minutes to train, as 100 epochs took 45 minutes to train. At 100 epochs the model reached an accuracy of 85.13%. The results for the model trained for 20 epochs are listed in Table 7.11.



(a) The positive and negative losses over 100 epochs.

(b) The positive and negative losses over 20 epochs.

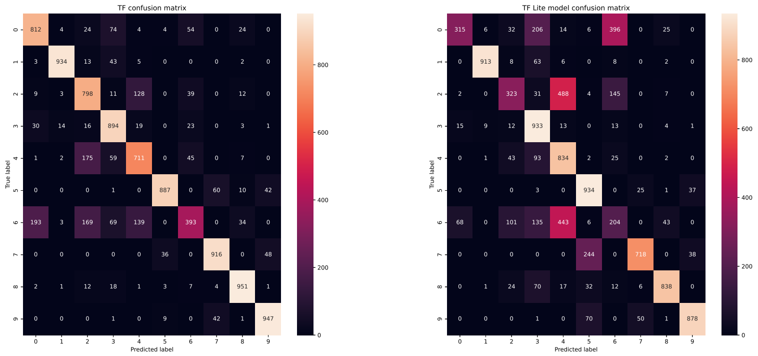
Figure 7.10: The positive and negative losses for 100 epochs (a) compared to 20 epochs (b).

	TF	TF Lite	MCU
Epochs trained	20	-	
Accuracy (%)	82.43	68.9	69.66
Training time	584.73	-	
Inference time per sample (μs)	604.1	8771.5	1065.4
Total model size (KiB)	-	26.8	184.83
Data buffer (KiB)	-	25.32	83.84
Non-data buffer (KiB)	-	1.48	101.99

Table 7.11: The results of the single-layer forward-forward model.

Comparing this model to backpropagation, a lot of differences can be observed. Forward-forward requires more epochs to train and each epoch takes longer to finish. The TF accuracy is almost 4 percentage points lower than backpropagation, but the TF Lite accuracies have dropped more than 15 points. This difference could be attributed to the effect of quantization, as the forward-forward model has not been trained as well as the backpropagation model when comparing the losses. The less trained forward-forward model could be more sensitive to the small perturbations of the weights as a result of quantization. This effect can be observed by looking at the confusion matrices shown in Fig. 7.11 Both of the matrices originate from the same TF model, but the result from the model converted to TF Lite is very different.

The last difference is the inference time. With how inference is performed, it is expected to have up to a 10 times increase in inference time. However the inference time for TF Lite on computer is more than 200 times higher compared to backpropagation.



(a) The confusion matrix of the TF model. **(b)** The confusion matrix of the TF Lite model.

Figure 7.11: The confusion matrix of the TF model (a) and the TF Lite model (b).

7.2.4. Small Single-Layer with Momentum

The single-layer forward-forward model was also tested with momentum. Momentum proved to significantly improve the training time of the standard single-layer backpropagation model. Learning rate decay and learning rate decay with momentum were not tested, as those did not show the same level of improvement over the standard single-layer model. The results are shown in Table 7.12. The accuracy has increased slightly, however the TF Lite accuracies decreased.

	TF	TF Lite	MCU
Epochs trained	20	-	
Accuracy (%)	83.1	68.73	66.83
Training time	457.15	-	
Inference time per sample (μs)	778.2	14222.3	1005.9
Total model size (KiB)	-	26.17	185.83
Data buffer (KiB)	-	24.72	83.84
Non-data buffer (KiB)	-	1.45	101.99

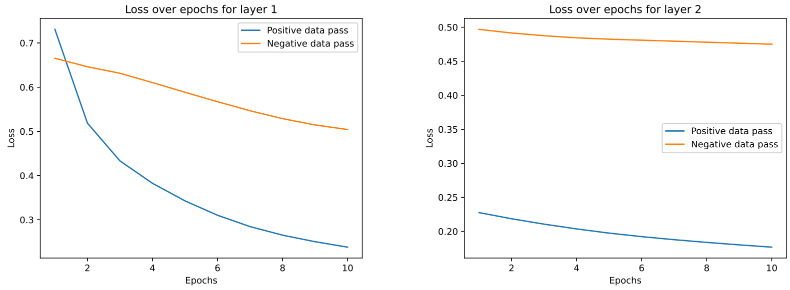
Table 7.12: The results of the single-layer model with momentum.

7.2.5. Small Two-Layer Network

This two-layer network is similar to the model discussed in Section 7.1.3. This model was trained for 10 epochs. The results of this network are listed in Table 7.13. The losses of the layers are plotted in Fig. 7.12. The losses of the first layer behave similar to that of the single-layer model, however, for the second layer the loss barely changes. The next two sections will investigate two ways to increase the loss reduction in the second layer.

	TF	TF first layer only	TF Lite	TF Lite first layer only	MCU
Epochs trained	10	10	-		
Accuracy (%)	79.46	79.61	59.79	72.10	52.87
Training time	570.06	-			
Inference time per sample (μs)	1374.0	619.8	16075.0	8173.4	1200.87
Total model size (KiB)	-		28.84	26.17	186.83
Data buffer (KiB)	-		25.95	24.72	84.84
Non-data buffer (KiB)	-		2.89	1.45	101.99

Table 7.13: The results of the small two-layer forward-forward model.



(a) The losses of the first layer. **(b)** The losses of the second layer.

Figure 7.12

7.2.6. Small Two-Layer with Variable Epochs

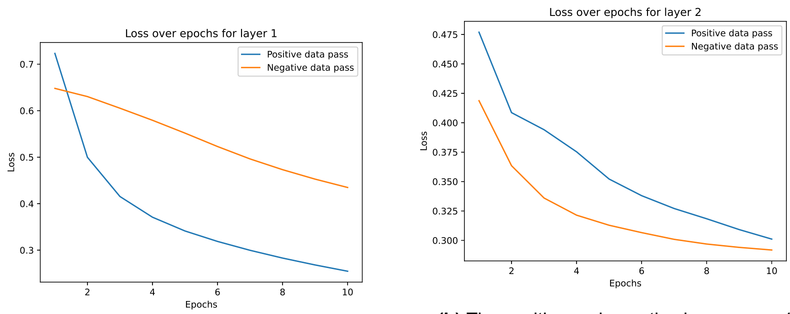
For this experiment the second layer of the two-layer model was trained with an additional 10 epochs. The layer configuration was unchanged from the two-layer network. For this test, the time limit was not enforced to isolate the effect of this. To enforce the time limit the epochs of the first layer would have to be reduced, this would obfuscate the effect of the additional epochs on the second layer. The results are listed in Table 7.14. Adding 10 epochs to the second layer did not significantly increase the performance of the model.

	TF	TF first layer only	TF Lite	TF Lite first layer only	MCU
Epochs trained	10-20	10	-		
Accuracy (%)	79.67	80.32	53.03	61.85	52.87
Training time	677.31	-			
Inference time per sample (μs)	1999.11	690.98	14472.08	29368.37	1200.87
Total model size (KiB)	-		28.84	26.17	186.83
Data buffer (KiB)	-		25.95	24.72	84.84
Non-data buffer (KiB)	-		2.89	1.45	101.99

Table 7.14: The results of the two-layer forward-forward model trained with 10 additional epochs for the second layer.

7.2.7. Small Two-Layer with Variable Learning Rate

Another way to reduce the loss in the second layer is to use a higher learning rate to train the second layer. The training parameters for the first layer remain unchanged, only the learning rate of the second layer was increased from 0.1 to 10. The model was trained for 10 epochs, similar to the model in Section 7.2.5. The losses of this model are plotted in 7.13, the results are listed in Table 7.15.



(a) The positive and negative losses over 10 epochs for the first layer with a learning rate of 0.1.

(b) The positive and negative losses over 10 epochs for the second layer with an adjusted learning rate of 10.

Figure 7.13: The positive and negative losses for the first layer with the unmodified learning rate (a) and for the second layer with the learning rate increased to 10 (b).

	TF	TF first layer only	TF Lite	TF Lite first layer only	MCU
Epochs trained	10	10	-		
Accuracy (%)	79.74	80.18	73.81	76.19	75.87
Training time		-			
Inference time per sample (μs)	3102.70	729.827	27784.14	13319.48	1200.42
Total model size (KiB)	-		28.84	26.17	186.83
Data buffer (KiB)	-		25.95	24.72	84.84
Non-data buffer (KiB)	-		2.89	1.45	101.99

Table 7.15: The results of the two-layer forward-forward model trained with a learning rate of 10 for the second layer.

7.2.8. Largest Single-Layer Model

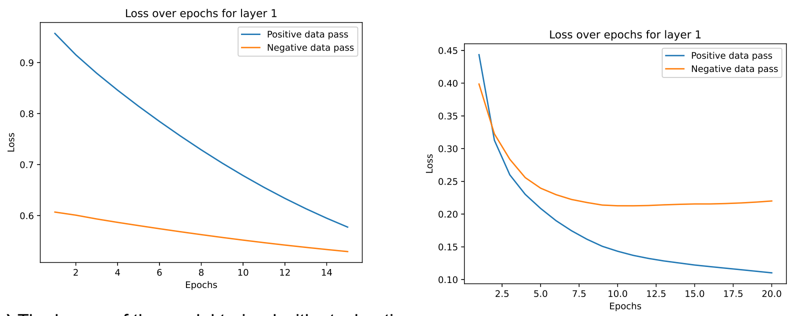
The largest single-layer model that was estimated to fit on the microcontroller consists of a single layer of 410 neurons. This model was trained with two different optimizers, Table 7.16 shows the model trained with stochastic gradient descent, Table 7.17 shows the same model trained with the Adam optimizer. The results show that the Adam optimizer achieved a higher accuracy, while training for less total time, even though it was trained with five more epochs.
The losses of both models are plotted in Fig. 7.14, these plots show that the Adam optimizer reduces the loss at a much faster rate compared to stochastic gradient descent.

	TF	TF Lite	MCU
Epochs trained	15	-	
Accuracy (%)	61.95	61.97	65.70
Training time	567.46	-	
Inference time per sample (μs)	1114.07	9554.99	7675.65
Total model size (KiB)	-	317.06	476.89
Data buffer (KiB)	-	315.61	374.84
Non-data buffer (KiB)	-	1.45	102.05

Table 7.16: The largest single-layer model that was estimated to fit on the MCU trained with stochastic gradient descent.

	TF	TF Lite	MCU
Epochs trained	20	-	
Accuracy (%)	85.64	85.66	86.85
Training time	474.20	-	
Inference time per sample (μs)	699.32	13447.59	7648.63
Total model size (KiB)	-	317.06	476.89
Data buffer (KiB)	-	315.61	374.84
Non-data buffer (KiB)	-	1.45	102.05

Table 7.17: The largest single-layer model that was estimated to fit on the MCU trained with the Adam optimizer.



(a) The losses of the model trained with stochastic gradient descent.

(b) The confusion matrix of the TF Lite model.

Figure 7.14: The losses of the model trained with the Adam optimizer.

7.2.9. Largest Two-layer Model

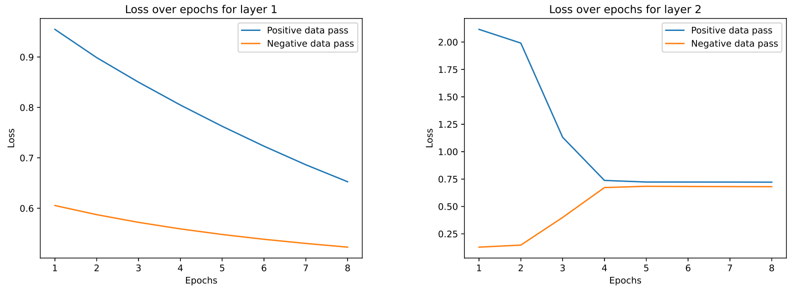
The largest two-layer model to fit in the MCU was estimated at two layers of 297 neurons. This model has also been trained with both stochastic gradient descent and the Adam optimizer. Table 7.18 shows the model trained with Stochastic gradient descent, Table 7.20 shows the model trained with the Adam optimizer. Comparing the losses in Fig. 7.15 and Fig. 7.16 shows again that the Adam optimizer minimizes the loss at a faster rate. Fig. 7.15b also shows an example where the negative losses are increased to lower the total losses.

	TF	TF first layer only	TF Lite	TF Lite first layer only	MCU
Epochs trained	8	8		-	
Accuracy (%)	62.54	60.70	62.43	61.30	64.24
Training time	581.91		-		
Inference time per sample (μs)	1451.01	1098.97	19458.95	9886.53	9714.12
Total model size (KiB)		-	318.96	230.09	476.89
Data buffer (KiB)		-	316.06	228.65	374.84
Non-data buffer (KiB)		-	2.90	1.44	102.05

Table 7.18: The results of the two-layer forward-forward model trained with stochastic gradient descent.

	TF	TF first layer only	TF Lite	TF Lite first layer only	MCU
Epochs trained	10	10		-	
Accuracy (%)	83.11	83.81	83.24	83.76	83.90
Training time	490.32		-		
Inference time per sample (μs)	1416.64	741.67	27784.14	13319.48	9590.32
Total model size (KiB)		-	318.94	230.09	476.89
Data buffer (KiB)		-	316.05	228.65	374.84
Non-data buffer (KiB)		-	2.89	1.44	102.05

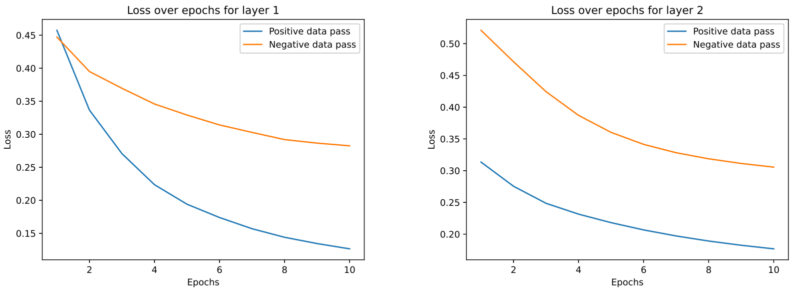
Table 7.19: The results of the two-layer forward-forward model trained with the Adam optimizer.



(a) The losses of the first layer of the model trained with stochastic gradient descent.

(b) The losses of the second layer of the model trained with stochastic gradient descent.

Figure 7.15: The losses of the model trained with stochastic gradient descent.



(a) The losses of the first layer of the model trained with the Adam optimizer.

(b) The losses of the second layer of the model trained with the Adam optimizer.

Figure 7.16: The losses of the model trained with the Adam optimizer plotted per layer.

7.2.10. Largest Three-Layer Model

The largest three-layer forward-forward model that fits on the MCU was estimated at three layers of 250 neurons. Table 7.20 shows the results of this model. This model is only shown trained with the Adam optimizer, Section 7.2.8 and 7.2.9 have already proven that for these models, stochastic gradient descent is inadequate. The training time is above the 10 minute limit, the amount of epochs to reach this limit was overestimated.

	TF			TF Lite			
	Full model	1 layer	2 layers	Full model	1 layer	2 layers	MCU
Epochs trained	10			-			
Accuracy (%)	81.03	81.77	81.22	81.13	81.82	81.22	82.54
Training time	644.20			-			
Inference time per sample (μs)	2629.48	869.67	1592.97	28332.14	9512.58	19753.60	10227.29
Total model size (KiB)	-			351.04	193.92	257.48	476.89
Data buffer (KiB)	-			346.70	192.48	254.59	373.84
Non-data buffer (KiB)	-			4.34	1.44	2.89	102.05

Table 7.20: The results of the two-layer forward-forward model trained with the Adam optimizer.

7.2.11. Summarizing the Forward-Forward Results

Compared to backpropagation, forward-forward needs more epochs to train and the time each epoch takes is also longer. Regarding performance, the accuracy of backpropagation for the same model topology is higher. The inference time is also much lower for backpropagation than for forward-forward. A forward-forward network can fit more neurons in the hidden layers compared to backpropagation, since it does not require an output layer, however this is canceled out on the MCU because it needs more memory dedicated to the code to run inference.

The best performing forward-forward model was the reference model, however this model was trained for 1000 epochs and does not fit on the MCU. The next best performing model was the largest single-layer model trained with Adam. However, the Adam optimizer is much more powerful than stochastic gradient descent, this cannot be compared to the backpropagation models, which have all been trained with stochastic gradient descent.
The best performing model using stochastic gradient descent was the small-single layer model trained with momentum with an accuracy of 83.1%. The best performing TF Lite model trained with stochastic gradient descent was the small single-layer model trained with a variable learning rate.

7.3. Hebbian Learning

For Hebbian learning only preliminary testing was done. The results claimed by [11] were reproduced and some models were made for testing. This implementation was chosen as it is accessible and it claimed to rival backpropagation in some scenarios. Because this was written in PyTorch, it would first have to be rewritten in TensorFlow in order to convert to TensorFlow Lite. However, it was decided not to proceed with Hebbian Learning. The main reason for this is time constraints, implementing on-device training would likely not be possible in time, it was also expected that converting the whole code to TensorFlow would lead to unexpected problems and delays. Another reason is that during preliminary testing, it was found that the results were not as good as backpropagation and forward-forward.
The results of the preliminary testing are listed in Table 7.21. These results have also been collected in Google Colab, however this was with the use of a GPU. The testing was done without modifications of the hyperparameters and with the use of Instar rule for Hebbian learning.
Other implementations were considered, however most that were found showed similar shortcomings. The most promising Hebbian learning implementation for this application was HebbNet [10]. HebbNet was claimed to achieve an accuracy of 93% on the MNIST dataset with the aid of thresholding and sparsity. However, HebbNet required a hidden layer of 2000 neurons and 200 epochs to train to achieve this result. HebbNet could not be tested however, the original implementation was not available for use and the attempts to reproduce it were unsuccessful.

Layer configuration	Test Accuracy (%)	Epochs trained	Training Time (s)
100-196-400	68.82	100	1421.38
32-32-32	51.20	20	294.07
50-50-50	54.39	20	300.71
100-100-100	62.52	20	302.95
500-500-500	69.36	40	1385.25

Table 7.21: Preliminary test results for Hebbian Learning.

7.4. Online Learning

A few models have also been trained on the MCU and the FPGA, the results of this are included in Appendix A. This will be referred to in Chapter 8.

8 Conclusion and Discussion

In all the tests that have been performed, backpropagation has proven to perform the best in all aspects that have been measured. The accuracy of backpropagation in both TensorFlow and TensorFlow Lite is higher than comparable forward-forward models, furthermore it trains faster, both in terms of epoch required to train and time per epoch. Lastly, it inferences faster. However, forward-forward has the capacity to rival backprop, in most cases it is only a few percentage points behind backpropagation.
The main problems found with forward-forward is that it takes significantly longer to train and multi-layered forward-forward can decrease performance compared to a single-layer forward-forward network. Another problem, specific to hardware applications, is that a sub-optimally trained forward-forward model is very sensitive to quantization. After conversion to a TensorFlow Lite model the accuracy dropped by more than 10 percentage points in multiple cases. Although, on-edge training resulted in similar accuracy as the model trained in TensorFlow.
The longer training time of forward-forward is the result of the positive and negative data requiring multiple passes through the network. This training process did not prove beneficial in this application, however it might perform better in a scenario with limited training data, where the benefit of making negative data is not negated by an abundance of positive data.
A potential benefit of forward-forward over backpropagation is that the layers are trained independently. This property was utilized in the forward-forward model with variable learning rate in Section 7.2.7. The ability to change training parameters per layer offers more possibilities in optimizing the training process compared to backpropagation.
Hebbian was not thoroughly investigated but it did not display the potential to rival backpropagation. The results from preliminary testing showed a performance of below 70% on the Fashion MNIST dataset on all tested configurations. This is 10 points below the requirement of 80%. This indicated that for smaller models for the MCU and FPGA, the performance would likely be even lower.

8.1. Future Research

There is still a lot of research that can be done following from what has been done in this paper:

- For some specific scenarios, the subsequent layer of a forward-forward trained model did improve accuracy. Which parameters and hyperparameters cause the subsequent layer of a multi-layered model to increase performance?
- The independent training of forward-forward allows any kind of processing to happen in between the layers. How would forward-forward perform when another model or black box was inserted between the layers?
- How could hyperparameters customized per layer affect the performance of the forward-forward algorithm?
- One of the biggest drawbacks of forward-forward is the training time. Can the training time of forward-forward be accelerated with specialized hardware or on specialized hardware?

References

[1] Sajad Alipour AmirReza Rajabi. *GitHub*. 2023. URL: https://github.com/amirrezarajabi/Tensorflow-Forward-Forward.

[2] Colby Banbury et al. "MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers". In: (Oct. 2020). URL: http://arxiv.org/abs/2010.11267.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT Press, 2005. ISBN: 0262541858.

[5] Josen Daniel De Leon and Rowel Atienza. "Depth Pruning with Auxiliary Networks for Tinyml". In: *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2022, pp. 3963–3967.

[6] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[7] Yale Embedded Systems Week (ESWEEK) Priyadarshini Panda. *ESWEEK 2021 Education - Spiking Neural Networks*. Nov. 2021. URL: https://www.youtube.com/watch?v=7TybET1CslM.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[9] Martin Görner. URL: https://twitter.com/martin_gorner/status/1599759770608549888/photo/1.

[10] Manas Gupta, ArulMurugan Ambikapathi, and Savitha Ramasamy. "HebbNet: A Simplified Hebbian Learning Framework to do Biologically Plausible Learning". In: *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2021, pp. 3115–3119. DOI: 10.1109/ICASSP39728.2021.9414241.

[11] Manas Gupta et al. "Is Bio-Inspired Learning Better than Backprop? Benchmarking Bio Learning vs. Backprop". In: (Dec. 2022). URL: https://arxiv.org/abs/2212.04614.

[12] Suyog Gupta et al. *Deep Learning with Limited Numerical Precision*. 2015. arXiv: 1502.02551 [cs.LG].

[13] D O Hebb. *The organization of behavior; a neuropsychological theory.* Wiley, 1949, pp. 335, xix, 335–xix.

[14] Geoffrey Hinton. URL: https://www.cs.toronto.edu/~hinton/.

[15] Geoffrey Hinton. "The forward-forward algorithm: Some preliminary investigations". In: *arXiv preprint arXiv:2212.13345* (2022).

[16] K. Fredrik Karlsson. "The Concept of Forward-Forward Learning Applied to a Multi Output Perceptron". In: (Apr. 2023). URL: http://arxiv.org/abs/2304.03189.

[17] Sindy Löwe. *GitHub*. 2020. URL: https://github.com/loeweX/Forward-Forward.

[18] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. "A History of Spike-Timing-Dependent Plasticity". In: *Frontiers in Synaptic Neuroscience* 3 (2011). ISSN: 1663-3563. DOI: 10.3389/fnsyn.2011.00004. URL: https://www.frontiersin.org/articles/10.3389/fnsyn.2011.00004.

[19] Martn Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[20] Thomas Miconi. *Hebbian learning with gradients: Hebbian convolutional neural networks with modern deep learning frameworks*. 2021. arXiv: 2107.01729 [cs.NE].

[21] Junho Yun Mohammad Pezeshki Haaris Rahman. *GitHub*. 2020. URL: https://github.com/mohammadpz/pytorch_forward_forward.

[22] Suvaditya Mukherjee. *Keras documentation: Using the forward-forward algorithm for Image Classification*. Aug. 2023. URL: https://keras.io/examples/vision/forwardforward/.

[23] Izaak Neutelings. URL: https://tikz.net/neural_networks/.

[24] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[25] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. "Machine Learning for Microcontroller-Class Hardware: A Review". In: *IEEE Sensors Journal* 22 (22 Nov. 2022), pp. 21362–21390. ISSN: 1530-437X. DOI: 10.1109/JSEN.2022.3210773. URL: https://ieeexplore.ieee.org/document/9912325/.

[26] Mingjia Shi et al. *Personalized Federated Learning with Hidden Information on Personalized Prior*. 2022. arXiv: 2211.10684 [cs.LG].

[27] Atsushi Takeda and Hanuna Tamano. "The Impact of Synaptic Zn2 Dynamics on Cognition and Its Decline". In: *International Journal of Molecular Sciences* 18.11 (Nov. 2017), p. 2411. DOI: 10.3390/ijms18112411. URL: https://doi.org/10.3390/ijms18112411.

[28] Richard Naud Werner M. Kistler and Liam Paninski Wulfram Gerstner. *Synaptic Plasticity and Learning*. URL: https://neuronaldynamics.epfl.ch/online/Ch19.html.

[29] Hao Wu et al. "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation". In: (Apr. 2020). URL: https://arxiv.org/abs/2004.09602v1.

[30] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017. arXiv: 1708.07747 [cs.LG].

[31] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[32] Kaichao You et al. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019. arXiv: 1908.01878 [cs.LG].

A

Measurements Of All Subgroups

<table>
<tr><td>BP</td><td colspan="5">Single layer 32</td><td colspan="5">Dual layer 32-32 !!!</td><td></td></tr>
<tr><td></td><td colspan="5">LR = 0.01, Batch size = 16</td><td colspan="5">LR = 0.01, Batch size = 16</td><td></td></tr>
<tr><td>Fashion MNIST</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>FPGA</td><td>TF Lite (MCU)</td><td>Local MCU</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU</td><td>FPGA</td></tr>
<tr><td>Datatype</td><td>fp32</td><td>int8</td><td>int8</td><td>int8</td><td>fp32</td><td>fp32</td><td>int8</td><td>int8</td><td>fp32</td><td>int8</td></tr>
<tr><td>Accuracy on test set after 1 epoch</td><td>80.72</td><td>–</td><td>–</td><td>–</td><td>79.99</td><td>81.03</td><td>–</td><td>–</td><td>80.06</td><td></td></tr>
<tr><td>Accuracy on test set after 5 epoch</td><td>85</td><td>–</td><td>–</td><td>–</td><td>84.54</td><td>85.07</td><td>–</td><td>–</td><td>85.69</td><td></td></tr>
<tr><td>Accuracy on test set after 10 epochs</td><td>85.82</td><td>–</td><td>–</td><td>–</td><td>85.63</td><td>86.66</td><td>–</td><td>–</td><td>86.84</td><td></td></tr>
<tr><td>Epochs trained</td><td>30</td><td>–</td><td>–</td><td>–</td><td>10</td><td>22</td><td>–</td><td>–</td><td>10</td><td></td></tr>
<tr><td>Accuracy for given epochs</td><td>86.41</td><td>86.33</td><td>85.53</td><td>86.33</td><td>–</td><td>87.46</td><td>87.36</td><td>87.36</td><td>–</td><td>85.67</td></tr>
<tr><td>Memory required (KiB) (Variable | Code)</td><td>–</td><td>25.68|1.34</td><td>25.08 | 1.34</td><td>60.16 | 105.74</td><td>265.22 | 99.90</td><td>–</td><td>26.20 | 2.33</td><td>61.16|105.74</td><td>278.47 | 102.02</td><td></td></tr>
<tr><td>Inference time per sample(us)</td><td>65.02</td><td>34.62</td><td>14.38</td><td>70.19</td><td>258.4</td><td>72.34</td><td>37.47</td><td>75.2</td><td>270.04</td><td>15.24</td></tr>
<tr><td>Training time (s)</td><td>142.2</td><td>–</td><td>-</td><td>–</td><td>372.9</td><td>123.9</td><td>–</td><td>–</td><td>396.6</td><td></td></tr>
<tr><td>Training time per epoch (s)</td><td>4.74</td><td>–</td><td>–</td><td>–</td><td>37.29</td><td>5.63</td><td>–</td><td>–</td><td>39.66</td><td></td></tr>
<tr><td>Energy / inference (mJ)</td><td>–</td><td>–</td><td>0.028</td><td>0.046</td><td>0.171</td><td>–</td><td>–</td><td>0.05</td><td>0.179</td><td>0.029</td></tr>
</table>

<table>
<tr><td>BP</td><td colspan="4">Single layer 32 + LR Decay</td><td colspan="4">Single layer 32 + Momentum + decay</td><td colspan="4">Single layer 32 + Momentum</td></tr>
<tr><td></td><td colspan="4">LR = 0.1, Every 200 batches LR = LR*0.95, Min LR = 0.01, Batch size = 16</td><td colspan="4">LR = 0.1, Every 200 batches LR = LR*0.95, Min LR = 0.01, Batch size = 16, Momentum = 0.9</td><td colspan="4">LR = 0.01, Batch size = 16</td></tr>
<tr><td>Fashion MNIST</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local (MCU)</td></tr>
<tr><td>Datatype</td><td>fp32</td><td>int8</td><td>int8</td><td>fp32</td><td>fp32</td><td>int8</td><td>int8</td><td>fp32</td><td>fp32</td><td>int8</td><td>int8</td><td>fp32</td></tr>
<tr><td>Accuracy on test set after 1 epoch</td><td>84.17</td><td>–</td><td>–</td><td>84.04</td><td>60.5</td><td>–</td><td>–</td><td>84.57</td><td>83.05</td><td>–</td><td>–</td><td>81.54</td></tr>
<tr><td>Accuracy on test set after 5 epoch</td><td>85.8</td><td>–</td><td>–</td><td>85.91</td><td>80.34</td><td>–</td><td>–</td><td>85.81</td><td>85.95</td><td>–</td><td>–</td><td>85.49</td></tr>
<tr><td>Accuracy on test set after 10 epochs</td><td>86.23</td><td>–</td><td>–</td><td>86.48</td><td>83.03</td><td>–</td><td>–</td><td>86.46</td><td>86.94</td><td>–</td><td>–</td><td>86.7</td></tr>
<tr><td>Epochs trained</td><td>26</td><td>–</td><td>–</td><td>10</td><td>76</td><td>–</td><td>–</td><td></td><td>12</td><td>–</td><td>–</td><td>10</td></tr>
<tr><td>Accuracy for given epochs</td><td>86.74</td><td>86.68</td><td>86.68</td><td>–</td><td>80.51</td><td>80.6</td><td>80.58</td><td></td><td>86.43</td><td>86.45</td><td>86.43</td><td>–</td></tr>
<tr><td>Memory required (KiB) (Variable | Code)</td><td>–</td><td>25.08 | 1.34</td><td>60.16|105.74</td><td>265.22 | 99.90</td><td></td><td>25.08 | 1.34</td><td>60.16|105.74</td><td>265.2 | 100.5</td><td>–</td><td>25.08 | 1.34</td><td>60.16|105.74</td><td>265.22 | 100.52</td></tr>
<tr><td>Inference time per sample(us)</td><td>86.97</td><td>30.55</td><td>70.1</td><td>258.4</td><td>74.21</td><td>38.99</td><td>69.99</td><td>??</td><td>69.1</td><td>24.83</td><td>70.35</td><td>258.3</td></tr>
<tr><td>Training time (s)</td><td>262.6</td><td>–</td><td>–</td><td>372.9</td><td>682.96</td><td>–</td><td>–</td><td>??</td><td>82.88</td><td>–</td><td>–</td><td>378.6</td></tr>
<tr><td>Training time per epoch (s)</td><td>10.1</td><td>–</td><td>–</td><td>37.29</td><td>8.99</td><td>–</td><td>–</td><td>??</td><td>6.91</td><td>–</td><td>–</td><td>37.86</td></tr>
<tr><td>Energy / inference (mJ)</td><td>–</td><td>–</td><td>0.046</td><td>0.199</td><td>–</td><td>–</td><td>–</td><td>??</td><td>–</td><td>–</td><td>0.047</td><td>0.171</td></tr>
</table>

<table>
<tr><td>FF</td><td colspan="5">Single layer 32</td><td>SL32</td><td>S. L. 104</td><td colspan="4">Dual layer 32-32 lr=0.1</td></tr>
<tr><td>Fashion MNIST</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU 32</td><td>Local MCU 16</td><td>FPGA quantized</td><td>FPGA</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU</td></tr>
<tr><td>Accuracy on test set after 1 epoch</td><td></td><td></td><td></td><td>62.02</td><td>69.32</td><td>0.6187</td><td>70.29</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Accuracy on test set after 5 epochs</td><td></td><td></td><td></td><td>76.68</td><td>82.26</td><td>0.6963</td><td>73.08</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Accuracy on test set after 10 epochs</td><td></td><td></td><td></td><td>80.9</td><td>83.74</td><td>0.6979</td><td>73.68</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Epochs trained</td><td>20</td><td></td><td></td><td>10</td><td>10</td><td>10</td><td>4</td><td>10</td><td>–</td><td>–</td><td>10</td></tr>
<tr><td>Total training time (s)</td><td>584.73</td><td></td><td></td><td></td><td></td><td></td><td></td><td>570.06</td><td>–</td><td>–</td><td>1063</td></tr>
<tr><td>On device training time per epoch (s)</td><td></td><td></td><td>–</td><td>71.18</td><td>71.18</td><td></td><td>5.07-6.16s/epoch</td><td>57.006</td><td>–</td><td>–</td><td>106.3</td></tr>
<tr><td>Accuracy after epochs trained</td><td>82.43</td><td>68.9</td><td>69.66</td><td></td><td>–</td><td>69.79</td><td>74.2</td><td>79.46</td><td>59.79</td><td>67.93</td><td>81.59</td></tr>
<tr><td>Memory required (kB) (Variable | Code)</td><td></td><td>25.32|1.48</td><td>57.06 | 105.18</td><td>318.19 | 98.09</td><td>267.06 | 98.09</td><td></td><td>256.5 (BRAM)????</td><td></td><td>26.58|2.96</td><td>70.47 | 105.18</td><td>327.31 | 99.21</td></tr>
<tr><td>Inference time per example(us) (all labels)</td><td>604.1</td><td>8771.5</td><td>1065.4</td><td>2533.27</td><td>2533.27</td><td></td><td>100.46</td><td>1374</td><td>16065</td><td>1189.1</td><td>2794.27</td></tr>
<tr><td>Energy / inference (mJ)</td><td></td><td></td><td>0.707</td><td>1.676</td><td>1.676</td><td></td><td>0.19</td><td>–</td><td>–</td><td>0.79</td><td>1.848</td></tr>
</table>

<table>
<tr><td>FF</td><td colspan="5">Dual layer 32-32 lr0.1 - 10</td><td colspan="4">Single layer 32 + momentum</td></tr>
<tr><td>Fashion MNIST</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU 32</td><td>Local MCU 16</td><td>TF (PC)</td><td>TF Lite (PC)</td><td>TF Lite (MCU)</td><td>Local MCU</td></tr>
<tr><td>Accuracy on test set after 1 epoch</td><td></td><td></td><td></td><td>59.52</td><td>70.02</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Accuracy on test set after 5 epochs</td><td></td><td></td><td></td><td>79.42</td><td>81.93</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Accuracy on test set after 10 epochs</td><td></td><td></td><td></td><td>81.21</td><td>82.06</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Epochs trained</td><td></td><td></td><td></td><td></td><td>10</td><td>20</td><td></td><td></td><td></td></tr>
<tr><td>Total training time (s)</td><td></td><td></td><td></td><td></td><td></td><td>457.15</td><td></td><td></td><td></td></tr>
<tr><td>On device training time per epoch (s)</td><td></td><td></td><td></td><td></td><td>106.3</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Accuracy after epochs trained</td><td></td><td></td><td></td><td></td><td></td><td>83.1</td><td>68.73</td><td>66.83</td><td></td></tr>
<tr><td>Memory required (kB) (Variable | Code)</td><td></td><td></td><td></td><td></td><td>274.19 | 99.21</td><td></td><td>25.32|1.48</td><td>185.83|101.99</td><td></td></tr>
<tr><td>Inference time per example(us) (all labels)</td><td></td><td></td><td></td><td></td><td>2794.27</td><td>778.2</td><td>14222</td><td>1005.9</td><td></td></tr>
<tr><td>Energy / inference (mJ)</td><td></td><td></td><td></td><td></td><td>1.848</td><td></td><td></td><td></td><td></td></tr>
</table>

B

Model Generation and Testing Notebooks

This appendix contains the notebooks that were developed during the project.

B.1. Backpropagation Notebook

This is the code used to generate, convert, and test models trained with backpropagation. The notebook was converted to a .py file for ease of reading.

```
# -*- coding: utf-8 -*-
"""bp_model_test.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/15wB5w-aohAvIT4rnjqSIOEmWyTHYOH8q
"""

import tensorflow as tf
from tensorflow import keras as keras
from keras.models import Sequential
from keras.layers import Dense, InputLayer
from keras.optimizers import Adam
from keras.datasets import mnist, fashion_mnist
from keras.utils import to_categorical
from keras.optimizers import SGD
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import numpy as np
import time
import os
import pickle
import seaborn as sn
import pandas as pd
from tensorflow.lite.python.util import convert_bytes_to_c_source

!unzip bp_model_32_200-30ep_lr0.01_b16.zip -d /content

!nvidia-smi
!cat /proc/cpuinfo

```

```python
TRAIN = False  # set to True to train the model, False to load the model from file

# hyperparameters
batch_size = 16
epochs = 200
learning_rate = 0.01
momentum = 0.0 #0.9

# dataset to use
dataset = {1: "mnist", 2: "fashion_mnist"}
# set the dataset to use
datanum = 2  # 1 for mnist, 2 for fashion mnist

layer_config = [40,20]  # number of neurons in the hidden layers [x,y, ...] = 784->x->y->...->10

# location to save the model
saved_model_folder = "Meta_data_and_data_models"

file_path = os.getcwd() # os.path.dirname(os.path.realpath(__file__))
saved_model_path = file_path+"/"+saved_model_folder+"/"

# if folder does not exist, create it
if not os.path.exists(saved_model_path):
    os.makedirs(saved_model_path)
    print("---------------------------------------------------------")
    print("Created directory: ", saved_model_path)
    print("---------------------------------------------------------")
    # write txt file with readme
    txt = """    In this folder will the models and meta model files be stored.
    regular model => .h5
    lite model => .tflite
    _his => training history information
    _time => average time of one epoch when training
    # base_name explained e.g: SGD_backprop_mnist_L32_B32_E5_LR0.01_M0.9
    # L32: 32 neurons in the hidden layer
    # B32: batch size 32
    # E5: 5 epochs
    # LR0.01: learning rate 0.01
    # M0.9: momentum 0.9"""
    with open(saved_model_path+"readme.txt", "w") as f:
        f.write(txt)


# base_name explained e.g: SGD_backprop_mnist_L32_B32_E5_LR0.01_M0.9
# L32: 32 neurons in the hidden layer
# B32: batch size 32
# E5: 5 epochs
# LR0.01: learning rate 0.01
# M0.9: momentum 0.9
base_name = "SGD_backprop_"+dataset[datanum]+"_" + str("".join([f"L{x}" for x in layer_config]))+"_BS"+str(batch_size)+"_E

# train is False if file exists
if os.path.isfile(saved_model_path+base_name) and os.path.isfile(saved_model_path+base_name+"_his.pickle") and os.path.isfi
    TRAIN = False
    print("---------------------------------------------------------")
    print("Configurations of the model found, loading model...")
    print("---------------------------------------------------------")
else:
```

```
    print("---------------------------------------------------------")
    print("Configurations of the model not found, training model...")
    print("---------------------------------------------------------")

# Load the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data() if datanum == 1 else fashion_mnist.load_data()

# Normalize pixel values to be between 0 and 1
train_images = train_images.astype(np.float32)/255
test_images = test_images.astype(np.float32)/255

train_images = train_images.reshape(-1, 784)
test_images = test_images.reshape(-1, 784)

# Convert labels to one-hot encoding
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# Define the model architecture
if TRAIN:
    model = Sequential([InputLayer(input_shape=(784,))]+[Dense(x, activation='relu') for x in layer_config]
                        +[Dense(10, activation='softmax')])
    # Compile the model
    model.compile(loss='categorical_crossentropy',
                  optimizer=SGD(learning_rate=learning_rate, momentum=momentum),
                  metrics=['accuracy'])
    model.summary()
    start = time.time()
    his = model.fit(train_images, train_labels, epochs=epochs, batch_size=batch_size,
                    validation_data=(test_images, test_labels))
    end = time.time()
    training_time = (end-start)
    # save model als pickle
    model.save(saved_model_path+base_name)
    # save history
    train_acc = his.history['accuracy']
    train_loss = his.history['loss']
    val_loss = his.history['val_loss']
    val_acc = his.history['val_accuracy']
    # save train_acc and val_acc
    pickle.dump([train_acc, val_acc], open(saved_model_path+base_name+"_acc.pickle", 'wb'))
    pickle.dump([train_loss, val_loss], open(saved_model_path+base_name+"_loss.pickle", 'wb'))
    pickle.dump(training_time, open(saved_model_path+base_name+"_time.pickle", 'wb'))

else:
    # load the model
    model = tf.keras.models.load_model(saved_model_path+base_name)
    # load the history
    train_acc, val_acc = pickle.load(open(saved_model_path+base_name+"_acc.pickle", 'rb'))
    train_loss, val_loss = pickle.load(open(saved_model_path+base_name+"_loss.pickle", 'rb'))
    training_time = pickle.load(open(saved_model_path+base_name+"_time.pickle", 'rb'))

#loss, tf_accuracy = model.evaluate(test_images, test_labels)

# first and fifth acc and loss of train and val
val_acc_e1 = val_acc[0]*100
val_acc_e5 = val_acc[5]*100
index_max_val = np.argmax(val_acc)
```

```python
index_max_train = np.argmax(train_acc)
max_val = val_acc[index_max_val]*100
max_train = train_acc[index_max_train]*100

print(f"\nMax training acc: {max_train}")
print(f"Max validation acc: {max_val}\n")

#plot loss over epochs
val_loss_idx = np.where(val_loss <= np.min(val_loss)*1.02)
train_loss_idx = np.where(train_loss <= np.min(train_loss)*1.02)
opt_epochs = train_loss_idx[train_loss_idx == val_loss_idx][0]


#plot accruacy over epochs
y_ax = range(1, epochs+1)
plt.plot(y_ax, train_loss, label='Training loss')
plt.plot(y_ax, val_loss, label='Validation loss')
plt.title('Loss over epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.ylim([np.min(train_loss) - 0.1, np.max(train_loss) +0.1])
plt.savefig(f"bp_loss_epochs_{base_name}.eps")
plt.show()
print(f"training loss within 2% of minimum at epoch {train_loss_idx[0][0]},
        validation loss {val_loss_idx[0][0]}")

# size of full model
tf_size = os.path.getsize(saved_model_path+base_name)
print(f"\nSize of full TF model: {tf_size} Bytes\n")

start = time.time()
tf_predictions = model.predict(test_images)
end = time.time()
tf_acc = sum(np.argmax(tf_predictions, axis=1) ==
             np.argmax(test_labels, axis=1))/len(test_images) * 100
tf_pred_time = end-start
tf_pred_time_sample = tf_pred_time/len(test_images)*1e6

print(f"\nModel acc: {tf_acc}%\nTotal inference time: {tf_pred_time} s\nPrediction time per sample: {tf_pred_time_sample} u
print(model.summary())

def convert_to_c(tflite_model, file_name):
    source, header = convert_bytes_to_c_source(tflite_model,  file_name)
    with  open(file_name + '.h',  'w')  as  h_file:
        h_file.write(header)
    with  open(file_name + '.cpp',  'w')  as  cpp_file:
        cpp_file.write(source)

# Convert the model to TensorFlow Lite format
if TRAIN:
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    # Apply post-training quantization
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    # quantize the weights to 8-bit integers
    converter.target_spec.supported_types = [tf.int8]

    def representative_data_gen():
```

```
        for input_value in tf.data.Dataset.from_tensor_slices(train_images).batch(1).take(100):
            yield [input_value]
    # provide a representative dataset to ensure we quantize correctly
    converter.representative_dataset = representative_data_gen
    converter.inference_input_type = tf.int8
    converter.inference_output_type = tf.int8

    # Convert the model
    tflite_model = converter.convert()
    # save the model
    with open(saved_model_path+base_name+".tflite", 'wb') as f:
        f.write(tflite_model)
    convert_to_c(tflite_model, saved_model_path + base_name)
else:
    tflite_model = open(saved_model_path+base_name+".tflite", 'rb').read()


# Analyze the tflite model
tf.lite.experimental.Analyzer.analyze(model_content=tflite_model)

# test inference on a images
# load the model
interpreter = tf.lite.Interpreter(model_path=saved_model_path+base_name+".tflite")
interpreter.allocate_tensors()

# time the inference
tfl_start = time.time()
tfl_predictions = []
tfl_acc = 0
for i, sample in enumerate(test_images):
    interpreter.set_tensor(interpreter.get_input_details()[0]['index'],
                           (sample*255 - 128).astype(np.int8).reshape(1, 784))
    interpreter.invoke()
    output = interpreter.get_tensor(interpreter.get_output_details()[0]['index'])
    output = np.argmax(output)
    tfl_predictions.append(output)

tfl_end = time.time()
tfl_pred_time = tfl_end-tfl_start

tfl_acc = np.sum(tfl_predictions == np.argmax(test_labels, axis=1))/len(test_labels)

# mean inference time in us
tfl_pred_time_sample = tfl_pred_time / len(test_labels) * 1e6

tfl_size = os.path.getsize(saved_model_path+base_name+".tflite")

print(f"TF Lite acc: {tfl_acc*100}%")
print(f"TF Lite inference time: {tfl_pred_time} s")
print(f"TF Lite inference time per sample: {tfl_pred_time_sample} us")

print("----------------------------------------------------------")
print("Tensorflow model:")
print(f"Accuracy: {tf_acc:.2f}%, time per image: {tf_pred_time_sample:.2f}us, size: {tf_size:.2f}B,
        Training time: {training_time:.2f}s")

# print(f"First epoch: Train acc: {first_train:.2f}%, Val acc: {first_val:.2f}%")
print(f"Max epoch: Train acc: {max_train:.2f}%, Val acc: {max_val:.2f}%")
```

```
print(f"index max epoch: Train acc: {index_max_train}, Val acc: {index_max_val}")

# get the labels
labels = np.argmax(test_labels, axis=1)

print("Classification report TF model")
print(classification_report(labels, np.argmax(tf_predictions, axis=1)))

# get the confusion matrix
tf_cm = confusion_matrix(labels, np.argmax(tf_predictions, axis=1))
tf_df_cm = pd.DataFrame(tf_cm, index = [i for i in range(10)],
                    columns = [i for i in range(10)])
plt.figure(figsize = (10,10))
sn.heatmap(tf_df_cm, annot=True, fmt='g')
plt.title('TF model confusion matrix')
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.savefig(f"tf_cm_{base_name}.eps")
plt.show()


print("----------------------------------------------------------")
print("Tensorflow Lite model:")
print(f"Accuracy: {tfl_acc*100:.2f}%, Time of one image: {tfl_pred_time_sample:.2f}ms,
        Size: {tfl_size:.2f}KB")

print("Classification report TF Lite model")
print(classification_report(labels, tfl_predictions))

tfl_cm = confusion_matrix(labels, tfl_predictions)
tfl_df_cm = pd.DataFrame(tfl_cm, index = [i for i in range(10)],
                    columns = [i for i in range(10)])
plt.figure(figsize = (10,10))
sn.heatmap(tfl_df_cm, annot=True, fmt='g')
plt.title('TF Lite model confusion matrix')
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.savefig(f"tfl_cm_{base_name}.eps")
plt.show()


print("----------------------------------------------------------")
print("Differences")
print(f"Accuracy: {tf_acc-tfl_acc*100:.2f}%, Time of one image:
        {tf_pred_time_sample-tfl_pred_time_sample:.2f}ms, Size: {tf_size-tfl_size:.2f}KB")

d_cm = np.abs(tfl_cm - tf_cm)
d_df_cm = pd.DataFrame(d_cm, index = [i for i in range(10)],
                    columns = [i for i in range(10)])
plt.figure(figsize = (10,10))
sn.heatmap(d_df_cm, annot=True, fmt='g')
plt.title('TF Lite - TF differences')
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.savefig(f"d_cm_{base_name}.eps")
plt.show()

print(val_acc_e1)
```

```
print(val_acc_e5)

!zip -r /content/bp_model_40-20_200-25ep_lr0.01_b16.zip /content/
```

B.2. Forward-Forward notebook

This is the code used to generate, convert, and test models trained with forward-forward. The notebook was converted to a .py file for ease of reading.

B.3. Backpropagation Teensy Implementation

```
# -*- coding: utf-8 -*-
"""ff_model_test.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1uKzSYca2N8vWdS3Fo0DzlMNBAQ2EaD3L
"""

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras as keras
from keras import datasets as kds
from keras import backend as K
from tqdm.auto import tqdm

import time
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.lite.python.util import convert_bytes_to_c_source
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

!nvidia-smi
!cat /proc/cpuinfo

TRAIN = True  # set to True to train the model, False to load the model from file

RECONVERT = False # covert trained model again to tflite else load tf lite model from file

UINT8 = False # set to True to convert to uint8 else float32

evaluate_per_layer = True # run tfl evaluation to analyse contribution per additional layer

export_figs = True

# hyperparameters
BATCH_SIZE = 32
EPOCHS = 20
LEARNING_RATE = 0.1
MOMENTUM = 0.0 #0.9
```

```

# dataset to use
DATASET = {1: "mnist", 2: "fashion_mnist"}
# set the dataset to use
DATANUM = 2 # 1 for mnist, 2 for fashion mnist

LAYER_CONFIG = [405] # number of neurons in the hidden layers [x,y, ...] = 784->x->y->...->10

nr_layers = len(LAYER_CONFIG)

# location to save the model
SAVED_MODEL_FOLDER = "Meta_data_and_data_models"

# helper functions
def edit_data(x, y, method="edit"):
    """Overlays the label on the image"""
    is_batch = x.ndim == 3
    if method == "edit":
        if is_batch:
            x[:, 0, :10] = 0.0
            for i in range(x.shape[0]):
                x[i, 0, y[i]] = 1.0
        else:
            x[0, :10] = 0.0
            x[0, y] = 1.0

def random_label(y):
    """Returns random label"""
    if type(y) != np.ndarray:
        label = list(range(10))
        del label[y]
        return np.random.choice(label)
    else:
        label = np.copy(y)
        for i in range(y.shape[0]):
            label[i] = random_label(y[i])
        return label

def FFLoss_with_threshold(threshold):
    def FFLoss(y_true, y_pred):
        g = K.pow(y_pred, 2)
        g = K.mean(g, axis=1)
        g = g - threshold
        g = g * y_true
        loss = K.log(1 + K.exp(g))
        loss = K.mean(loss)
        return loss
    return FFLoss

def normalize_FF(x_):
    """Normalize layer input"""
    return x_ / (tf.norm(x_, ord=2, axis=1, keepdims=True) + 1e-4)

def normalize_FF_np(x):
    """Normalize layer input using numpy isntead of tf"""
    return x / (np.linalg.norm(x.astype(float), ord=2, axis=1, keepdims=True) + 1e-4)

def train_layer(layer, batch_size, nr_epochs, pos, neg):
    """Train one layer of the FF model with positive and negative data"""
```

```python
    y_pos = np.ones(batch_size) * -1
    y_neg = np.ones(batch_size)
    pos_loss = []
    neg_loss = []
    for ep in tqdm(range(nr_epochs)):
        for b in range(pos.shape[0] // batch_size):
            x = pos[b * batch_size: (b + 1) * batch_size]
            pos_res = layer.train_on_batch(x, y_pos)
            x = neg[b * batch_size: (b + 1) * batch_size]
            neg_res = layer.train_on_batch(x, y_neg)
        pos_loss.append(pos_res)
        neg_loss.append(neg_res)
    return pos_loss, neg_loss

def train_model(layer_list, x_train, y_train, batch_size, nr_epochs):
    """Train the whole FF model"""
    x_pos, x_neg = make_pos_neg(x_train, y_train)
    pos_losses = []
    neg_losses = []
    for i, layer in enumerate(layer_list):
        #layer_list[i] =
        pos_loss, neg_loss = train_layer(layer, batch_size, nr_epochs, x_pos, x_neg)
        if i != (len(layer_list) - 1):
            x_pos = layer_list[i].predict(x_pos)
            x_neg = layer_list[i].predict(x_neg)
            x_pos = normalize_FF(x_pos)
            x_neg = normalize_FF(x_neg)
        pos_losses.append(pos_loss)
        neg_losses.append(neg_loss)
    return pos_losses, neg_losses

def make_model(dims, loss_threshold=2, optimizer=tf.keras.optimizers.legacy.Adam()):
    """Contruct a FF model"""
    model_layers = []
    if (len(dims) - 1) > 0:
        new_layer = tf.keras.Sequential([
            tf.keras.layers.Dense(dims[1],
                                  activation="relu",
                                  input_shape=[dims[0]])],
            name=f'layer_1')
        model_layers.append(new_layer)
    for d in range(len(dims) - 2):
        new_layer = tf.keras.Sequential([
            tf.keras.layers.Dense(dims[d + 2],
                                  activation="relu",
                                  input_shape=[dims[d + 1]])],
            name=f'layer_{d + 2}')
        model_layers.append(new_layer)

    optimizer = optimizer

    # Compile the model layers
    for layer in model_layers:
        layer.compile(loss=FFLoss_with_threshold(loss_threshold), optimizer=optimizer)
    return model_layers

def predict_sample(z, layers):
    """Returns prediction for a single sample"""
```

```
    z = z.reshape(1, 784)
    zs = [np.copy(z) for _ in range(10)]
    ans = 0

    for i in range(10):
        edit_data(zs[i], i)
    for i, layer in enumerate(layers):
        zs = [layer.predict(zs[i], verbose=0) for i in range(10)]
        ans += np.array([np.mean(np.power(zs[i], 2)) for i in range(10)])
        zs = [normalize_FF(zs[i]) for i in range(10)]
    return np.argmax(ans)

def test_FF_model(z, layers):
    """Runs prediction on a set of samples, returns the predicted label"""
    anses = []

    for i in tqdm(range(10)):
        tmp = np.copy(z)
        edit_data(tmp, np.ones((tmp.shape[0]), dtype=int) * i)
        tmp = tmp.reshape(tmp.shape[0], -1)
        ans = 0
        for layer in layers:
            tmp = layer.predict(tmp, verbose=0)
            ans += np.mean(np.power(tmp, 2), axis=1)
            tmp = normalize_FF(tmp)
        anses.append(ans.reshape(-1, 1))
    ans = np.concatenate(anses, axis=1)
    return np.argmax(ans, axis=1)

def make_pos_neg(x_train, y_train):
    pos = np.copy(x_train)
    neg = np.copy(x_train)
    edit_data(pos, y_train)
    edit_data(neg, random_label(y_train))
    pos = pos.reshape(pos.shape[0], -1)
    neg = neg.reshape(neg.shape[0], -1)
    return pos, neg

def convert_to_c(tflite_model, file_name):
    source, header = convert_bytes_to_c_source(tflite_model,  file_name)
    with  open(file_name + '.h',  'w')  as  h_file:
        h_file.write(header)
    with  open(file_name + '.cpp',  'w')  as  cpp_file:
        cpp_file.write(source)

def conv(model, save_path):
    tflite_models = []
    interpreters = []

    def representative_data_gen():
      for input_value in tf.data.Dataset.from_tensor_slices(train_images).batch(1).take(100):
        yield[input_value]

    for i, layer in enumerate(model):
      converter = tf.lite.TFLiteConverter.from_keras_model(model[i])
      converter.optimizations = [tf.lite.Optimize.DEFAULT]
      converter.target_spec.supported_types = [tf.int8]
      converter.representative_dataset = representative_data_gen
```

```python

        tflite_model = converter.convert()
        tflite_models.append(tflite_model)

        tf.lite.experimental.Analyzer.analyze(model_content=tflite_model)

        with open(f"{save_path}{i}.tflite", 'wb') as f:
          f.write(tflite_model)

        interpreter = tf.lite.Interpreter(model_path= (f"{save_path}{i}.tflite"))
        interpreters.append(interpreter)
        path = f"{save_path}_{i}"
        convert_to_c(tflite_model, path)
    return tflite_models, interpreters

def evaluate(model, x, y, nr_layers, datatype=0):
    # Run predictions on every image in the "test" dataset.
    prediction_digits = []
    x = x.reshape(x.shape[0], -1)

    interpreters = model[0:nr_layers]
    print("Evaluating tfl model")
    start = time.time()
    for j, sample in enumerate(x):
        # Pre-processing: add batch dimension and convert to uint8 to match with
        # the model's input data format.

        activation = []
        for i in range(10):
            # Test all labels
            if datatype == 0:
              pic_labeled = np.zeros((1, 784)).astype(np.float32)
            else:
              pic_labeled = np.zeros((1, 784)).astype(np.uint8)

            pic_labeled[0, :] = sample.copy()
            pic_labeled[0, 0:10] = 0.0
            pic_labeled[0, i] = 1.0
            res = 0
            for interpreter in interpreters:
                input_index = interpreter.get_input_details()[0]["index"]
                output_index = interpreter.get_output_details()[0]["index"]

                interpreter.allocate_tensors()
                interpreter.set_tensor(input_index, pic_labeled)
                # Run inference
                interpreter.invoke()

                # Post-processing: remove batch dimension and find the digit with highest
                # probability.
                layer_output = interpreter.get_tensor(output_index)
                res += np.mean(np.power(layer_output[0], 2))
                pic_labeled = normalize_FF(layer_output)

            activation.append(res)
        prediction_digits.append(np.argmax(activation))

    # Compare prediction results with ground truth labels to calculate accuracy.
    accurate_count = 0
```

```
    for index in range(len(prediction_digits)):
        if prediction_digits[index] == y[index]:
            accurate_count += 1
    accuracy = accurate_count * 1.0 / len(prediction_digits)
    end = time.time()
    ev_time = end-start
    print(f"time per sample tfl: {ev_time/len(x)}")
    return accuracy, prediction_digits

def metrics(x, y, model, tfl_models, tfl_interp, model_path, base_name, export_figs=False):
    class_names = [i for i in range(10)]
    start = time.time()
    predictions = test_FF_model(x, model)
    end = time.time()
    pred_time = end-start
    print(f"time per sample: {pred_time/len(x)}")
    # evaluate the normal model, confusion matrix, f-1, precision, recall
    cm = confusion_matrix(y, predictions)
    df_cm = pd.DataFrame(cm, index = [i for i in class_names],
                        columns = [i for i in class_names])
    plt.figure(figsize = (10,10))
    sn.heatmap(df_cm, annot=True, fmt='g')
    plt.title('TF Lite confusion matrix')
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
    if export_figs:
      plt.savefig(f"tf_cm_{base_name}.eps")
    plt.show()
    print("Classification report for TF model")
    print(classification_report(y, predictions))

    # evaluate tf lite model
    tfl_acc, tfl_pred = evaluate(tfl_interp, x, y, len(tfl_models))
    print(len(tfl_pred))
    print(len(y.shape))
    cm_lite = confusion_matrix(y, tfl_pred)
    df_cm_lite = pd.DataFrame(cm_lite, index = [i for i in class_names],
                        columns = [i for i in class_names])
    plt.figure(figsize = (10, 10))
    sn.heatmap(df_cm_lite, annot=True, fmt='g')
    plt.title('TF Lite model confusion matrix')
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
    if export_figs:
      plt.savefig(f"tfl_cm_{base_name}.eps")
    plt.show()
    print("Classification report for TF Lite model.eps")
    print(classification_report(y, tfl_pred))

    # confusion matrix with the differnces
    cm_delta = np.abs(cm_lite - cm)
    df_cm = pd.DataFrame(cm_delta, index = [i for i in class_names],
                        columns = [i for i in class_names])
    plt.figure(figsize = (10,10))
    sn.heatmap(df_cm, annot=True, fmt='g')
    plt.title('Confusion matrix with the differnces')
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
```

```
    if export_figs:
      plt.savefig(f"d_cm_{base_name}.eps")
    plt.show()

    train_acc = 0
    test_acc = 0

    train_pred = test_FF_model(train_images, model)
    test_pred = test_FF_model(test_images, model)

    train_acc = np.sum(train_pred == train_labels)
    test_acc = np.sum(test_pred == test_labels)

    train_acc = train_acc / train_labels.shape[0] * 100
    test_acc = test_acc / test_labels.shape[0] * 100
    # for tfl_model in tfl_models:
    print("Tensorflow model accuracy:", test_acc)
    # compare with original model
    print("Tensorflow lite model accuracy:", tfl_acc*100)
    # size of the model in bytes
    model_size = 0
    for i in range(nr_layers):
      model_size += os.path.getsize(f"{model_path}{i}.tflite")
    print(f"Model size: {model_size} bytes")

def import_tf_model(nr_layers, base_name, custom_objects):
    model = []
    for i in range(nr_layers):
        tf_layer = tf.keras.models.load_model(f"{base_name}{i}", custom_objects = custom_objects)
        model.append(tf_layer)
    return model

def import_tfl_model(nr_layers, base_name):
    model = []
    interpreters = []
    for i in range(nr_layers):
        tfl_model = open(f"{base_name}{i}.tflite")
        model.append(tfl_model)
        interpreter = tf.lite.Interpreter(model_path= (f"{base_name}{i}.tflite"))
        interpreters.append(interpreter)
        nr_layers = i+1
    return model, interpreters, nr_layers

FILE_PATH = os.getcwd() # for jupyter notebook

SAVED_MODEL_PATH = FILE_PATH+"/"+SAVED_MODEL_FOLDER+"/"

# BASE_NAME explained e.g: SGD_FF_mnist_L32_B32_E5_LR0.01_M0.9
# L32: 32 neurons in the hidden layer
# B32: batch size 32
# E5: 5 epochs
# LR0.01: learning rate 0.01
# M0.9: momentum 0.9

BASE_NAME= "SGD_FF_"+DATASET[DATANUM]+"_"+ str("".join([f"L{x}"for x in LAYER_CONFIG]))+"_BS"+str(BATCH_SIZE)+"_E"+str(EPOC

model_path = SAVED_MODEL_PATH + BASE_NAME
```

```python
# # Loading MNIST data

if os.path.isfile(SAVED_MODEL_PATH+BASE_NAME) and os.path.isfile(SAVED_MODEL_PATH+BASE_NAME+".tflite"):
    TRAIN = False
    print("----------------------------------------------------------")
    print("Configurations of the model found, loading model...")
    print("----------------------------------------------------------")
else:
    print("----------------------------------------------------------")
    print("Configurations of the model not found, training model...")
    print("----------------------------------------------------------")

# load data
(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data() if DATANUM == 1 else keras.data

# normalize
train_images = train_images.astype(np.float32)/255
test_images = test_images.astype(np.float32)/255

# convert to int8
train_images_int8 = train_images.astype(np.uint8)
test_images_int8 = test_images.astype(np.uint8)

if TRAIN:
    models = make_model([784] + LAYER_CONFIG, loss_threshold=2) # , optimizer=tf.keras.optimizers.legacy.SGD(learning_rate=

    for model in models:
      model.summary()
    start = time.time()
    pos_loss, neg_loss = train_model(models, train_images, train_labels, BATCH_SIZE, EPOCHS)
    end = time.time()

    with open("pos_losses", "w") as fout:
      fout.write(','.join(str(i) for i in pos_loss))
    with open("neg_losses", "w") as fout:
      fout.write(','.join(str(i) for i in neg_loss))

    y_ax = range(1, EPOCHS+1)
    for i in range(nr_layers):
      plt.plot(y_ax, pos_loss[i], label='Positive data pass')
      plt.plot(y_ax, neg_loss[i], label='Negative data pass')
      plt.title(f"Loss over epochs for layer {i+1}")
      plt.xlabel('Epochs')
      plt.ylabel('Loss')
      plt.legend()
      if export_figs:
        plt.savefig(f"loss_epoch_{BASE_NAME}.eps")
      plt.show()
    total_time = end-start
    print(f"Training time: {end-start:.2f}s")

    # save model
    for i, model in enumerate(models):
      model.save(f"{SAVED_MODEL_PATH}{BASE_NAME}{i}")

    train_acc = 0
    test_acc = 0
```

```
    print("Full model")
    print("Predicting training data")
    train_pred = test_FF_model(train_images, models)
    print("Predicting test data")
    test_pred = test_FF_model(test_images, models)
    train_acc = np.sum(train_pred == train_labels)
    test_acc = np.sum(test_pred == test_labels)
    train_acc = train_acc / train_labels.shape[0] * 100
    test_acc = test_acc / test_labels.shape[0] * 100
    print(f"train acc : {train_acc:.2f}%")
    print(f"test acc : {test_acc:.2f}%")

    # print("Only first layer")
    # print("Predicting training data")
    # train_pred_1 = test_FF_model(train_images, [models[0]])
    # print("Predicting test data")
    # test_pred_1 = test_FF_model(test_images, [models[0]])
    # train_acc_1 = np.sum(train_pred_1 == train_labels)
    # test_acc_1 = np.sum(test_pred_1 == test_labels)
    # train_acc_1 = train_acc_1 / train_labels.shape[0] * 100
    # test_acc_1 = test_acc_1 / test_labels.shape[0] * 100
    # print(f"train acc layer 1 : {train_acc_1:.2f}%")
    # print(f"test acc layer 1: {test_acc_1:.2f}%")

    # print("Only first and second layer")
    # print("Predicting training data")
    # train_pred_2 = test_FF_model(train_images, models[0:2])
    # print("Predicting test data")
    # test_pred_2 = test_FF_model(test_images, models[0:2])
    # train_acc_2 = np.sum(train_pred_2 == train_labels)
    # test_acc_2 = np.sum(test_pred_2 == test_labels)
    # train_acc_2 = train_acc_2 / train_labels.shape[0] * 100
    # test_acc_2 = test_acc_2 / test_labels.shape[0] * 100
    # print(f"train acc layer 2 : {train_acc_2:.2f}%")
    # print(f"test acc layer 2: {test_acc_2:.2f}%")

    tfl_models, interpreters = conv(models, model_path)
else:
    custom_objects = {"FFLoss": FFLoss_with_threshold}
    models = import_tf_model(nr_layers, model_path, custom_objects)
    tfl_models, interpreters, tfl_layers = import_tfl_model(nr_layers, model_path)
    print(tfl_layers)

    with open("pos_losses", "r") as fin:
      line = fin.readline()
      pos_loss = line.split(",")
    with open("neg_losses", "r") as fin:
      line = fin.readline()
      neg_loss = line.split(",")

    print("Predicting training data")
    train_pred = test_FF_model(train_images, models)
    print("Predicting test data")
    test_pred = test_FF_model(test_images, models)

    train_acc = np.sum(train_pred == train_labels)
    test_acc = np.sum(test_pred == test_labels)

```

```
    train_acc = train_acc / train_labels.shape[0] * 100
    test_acc = test_acc / test_labels.shape[0] * 100

    print(f"train acc : {train_acc:.2f}%")
    print(f"test acc : {test_acc:.2f}%")

metrics(test_images, test_labels, models, tfl_models, interpreters, model_path, BASE_NAME, export_figs=True)

# evaluate effects of each layer
if evaluate_per_layer:
  for i in range(nr_layers):
      tfl_acc, tfl_pred = evaluate(interpreters, test_images, test_labels, i+1)
      print(f"accuracy : {tfl_acc*100:.2f}%")

!zip -r /content/ff_model_405_20.zip /content/
```

C Teensy Tensorflow Lite Implementation

C.1. Backpropagation Teensy Implementation

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_log.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/micro/micro_utils.h"
#include "constants.h"
#include "model.h"
#include "output_handler.h"
#include <TensorFlowLite.h>
#include <Arduino.h>
// #include <SPI.h>
#include <SD.h>
#include <cstdint>

#define NN_INPUT_SIZE 784

#define NN_OUTPUT_SIZE 10
constexpr int kTensorArenaSize = 1024 * 10;
// Keep aligned to 16 bytes for CMSIS
alignas(16) uint8_t tensor_arena[kTensorArenaSize];

class tf{
  public:

    const tflite::Model* mod = nullptr;
    tflite::MicroInterpreter* interpreter = nullptr;
    // tflite::MicroMutableOpResolver<5> resolver;
    TfLiteTensor* input = nullptr;
    TfLiteTensor* output = nullptr;
    TfLiteStatus allocate_status;
    TfLiteStatus invoke_status;

    float theta = 2.0;

    int input_size;
```

```
    int output_size;
    int label = 0;

    tf(const unsigned char model[], int input_size, int output_size){
      input_size = input_size;
      output_size = output_size;

      tflite::InitializeTarget();

      mod = tflite::GetModel(model);
      if (mod->version() != TFLITE_SCHEMA_VERSION) {
        Serial.printf("Model schema version %d is not compatible with supported version %d\n",
                      mod->version(), TFLITE_SCHEMA_VERSION);
        return;
      }
      static tflite::MicroMutableOpResolver<5> resolver;

      resolver.AddQuantize();
      resolver.AddDequantize();
      resolver.AddFullyConnected();
      resolver.AddRelu();
      resolver.AddSoftmax();

      static tflite::MicroInterpreter static_interpreter(mod, resolver, tensor_arena, kTensorArenaSize);
      interpreter = &static_interpreter;

      allocate_status = interpreter->AllocateTensors();
      if (allocate_status != kTfLiteOk) {
        Serial.println("AllocateTensors() failed");
        return;
      }


      Serial.printf("Optimal tensor arena size: %d\n", interpreter->arena_used_bytes());
      // Obtain pointers to the model's input and output tensors.
      input = interpreter->input(0);
      if (input->type == kTfLiteInt8) {
        Serial.println("Model input type is kTfLiteInt8");
      } else if (input->type == kTfLiteFloat32) {
        Serial.println("Model input type is kTfLiteFloat32");
      } else if (input->type == kTfLiteUInt8) {
        Serial.println("Model input type is kTfLiteUInt8");
      } else if (input->type == kTfLiteInt16) {
        Serial.println("Model input type is kTfLiteInt16");
      } else if (input->type == kTfLiteInt32) {
        Serial.println("Model input type is kTfLiteInt32");
      } else {
        Serial.println("Model input type is unknown");
      }
      Serial.println("Model loaded successfully");

      // print dimensions of input tensor
      Serial.printf("Input tensor dimension count: %d\n", input->dims->size);
      Serial.printf("Input tensor dimensions: {");
      for (int i = 0; i < input->dims->size; i++) {
        Serial.printf("%d", input->dims->data[i]);
        if (i < input->dims->size - 1) {
          Serial.printf(", ");
```

```
      }
    }
    Serial.printf("}\n");

    output = interpreter->output(0);
    if (output->type == kTfLiteInt8) {
      Serial.println("Model output type is kTfLiteInt8");
    } else if (output->type == kTfLiteFloat32) {
      Serial.println("Model output type is kTfLiteFloat32");
    } else if (output->type == kTfLiteUInt8) {
      Serial.println("Model output type is kTfLiteUInt8");
    } else if (output->type == kTfLiteInt16) {
      Serial.println("Model output type is kTfLiteInt16");
    } else if (output->type == kTfLiteInt32) {
      Serial.println("Model output type is kTfLiteInt32");
    } else {
      Serial.println("Model output type is unknown");

    }
    // print dimensions of output tensor
    Serial.printf("Output tensor dimension count: %d\n", output->dims->size);
    Serial.printf("Output tensor dimensions: {");
    for (int i = 0; i < output->dims->size; i++) {
      Serial.printf("%d", output->dims->data[i]);
      if (i < output->dims->size - 1) {
        Serial.printf(", ");
      }
    }
    Serial.printf("}\n");


  }


float* get_output(){
  switch (output->type) {
    case kTfLiteFloat32:
      return output->data.f;
    case kTfLiteInt8:
      return (float*)output->data.int8;
    case kTfLiteUInt8:
      return (float*)output->data.uint8;
  }

}
void set_input(float* input_data){

  for (int i = 0; i < this->input_size; i++) {
    switch (input->type) {
      case kTfLiteFloat32:
        input->data.f[i] = input_data[i];
        break;
      case kTfLiteInt8:
        input->data.int8[i] = (int8_t)input_data[i];
        break;
      case kTfLiteUInt8:
        input->data.uint8[i] = (uint8_t)input_data[i];
        break;
```

```
    }
    }
  }
  float* get_input(){
    // only callable after invoking inference
    switch (input->type)
    {
    case kTfLiteFloat32:
      return input->data.f;
    case kTfLiteInt8:
      // fixed point to float
      return (float*) input->data.int8;
    case kTfLiteUInt8:
      return (float*) input->data.uint8;
    }
  }

  void inference(){

      TfLiteStatus invoke_status = interpreter->Invoke();
      if (invoke_status != kTfLiteOk) {
        Serial.printf("Invoke failed\n");
        return;
      }
      output = interpreter->output(0);
    }

};

void normalize(float* input_data, int input_size){
  float sum = 0;
  for (int i = 0; i < input_size; i++) {
    sum += input_data[i];
  }
  for (int i = 0; i < input_size; i++) {
    input_data[i] = input_data[i] / sum;
  }
}

int argmax( int input_size, TfLiteTensor* output){
  int max_index = 0;
  float max_value = 0;
  float value = -1000;
  for (int i = 0; i < input_size; i++) {
    switch (output->type) {
      case kTfLiteFloat32:
        value = output->data.f[i];
        break;
      case kTfLiteInt8:
        value = (float) (output->data.int8[i] - output->params.zero_point) * output->params.scale;
        break ;
      case kTfLiteUInt8:
        value= (output->data.uint8[i]  - output->params.zero_point) * output->params.scale;
        break;

    }
    if (value > max_value) {
      max_value = value;
```

```
      max_index = i;
    }

  }
  return max_index;
}


int label = 0;

int read_example(File &file, TfLiteTensor *input) {
  // float pic[NN_INPUT_SIZE];
  if (!file) {
    Serial.println("File reading failed!");
    return;
  }

  char buffer[NN_INPUT_SIZE + 1];
  file.readBytes(buffer, NN_INPUT_SIZE+1); // Read a single label + example into buffer
  label = (int) (uint8_t) buffer[0];
  // Scale, quantize and copy data from buffer to input tensor
  for (int i = 0; i < NN_INPUT_SIZE; i++) {
    switch (input->type) {
      case kTfLiteFloat32:
        input->data.f[i] = ((float) (uint8_t) buffer[i+1]) / 255.0;
        break;
      case kTfLiteInt8:
        input->data.int8[i] = (int8_t)(buffer[i+1] - 128);//tflite::FloatToQuantizedType<int8_t>(((float) (uint8_t) buffer[
        break;
      case kTfLiteUInt8:
        input->data.uint8[i] = (uint8_t)buffer[i+1];//tflite::FloatToQuantizedType<uint8_t>(((float) (uint8_t) buffer[i+1])
        break;
    }
  }
  return label;
}


void encode_label(float *input_data, int label) {
  for (int i=0; i < 10; i++) {
    input_data[i] = label == i ? 1.0 : 0.0;
  }
}

void setup() {
  // tflite::InitializeTarget();
  srand(millis());

  if (!SD.begin(BUILTIN_SDCARD)) {
    Serial.println("SD card initialization failed!");
    while(true);
  }

  delay(5000);

}

void print_pic(float* pic){
```

```
  for (int i = 0; i < NN_INPUT_SIZE; i++) {
    if (pic[i] > 0.01) {
      Serial.print("X");
    } else {
      Serial.print(" ");
    }
    if ((i+1) % 28 == 0) {
      Serial.println();
    }
  }
}

tf tf_model(g_model, NN_INPUT_SIZE, NN_OUTPUT_SIZE);
void loop() {
  // Ask for mnist image input
  Serial.println("Starting inference on test set");
  File testFile = SD.open("fashion_mnist_test.bin");
  int correct = 0;
  int total = 0;
  float tot_time = 0;
  // Serial.println("start");

  // Serial.println("model loaded");

  while (testFile.available()) {
    // Get a new entry from the file
    label = read_example(testFile, tf_model.input);
    float begin = micros();
    tf_model.inference();
    int predicted_label = argmax( NN_OUTPUT_SIZE, tf_model.output);
    float end = micros();
    tot_time += (end - begin);
    if (predicted_label == label) {
      correct ++;
      }
    total ++;
  }
  Serial.printf("------------------------------------------------------\n");
  Serial.printf("SGD_backprop_fashion_mnist_L32_BS16_E76_LR0.1_M0.9\n");
  Serial.printf("Accuracy: %f\n", ((float) correct) / (float) total);
  Serial.printf("Average inference time (ms): %f\n", tot_time /total );
}
```

C.2. Forward-Forward Teensy Implementation

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_log.h"
#include "tensorflow/lite/micro/system_setup.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/micro/micro_utils.h"
#include "constants.h"
#include "model.h"
#include "output_handler.h"
#include <TensorFlowLite.h>
#include <Arduino.h>
```

```
// #include <SPI.h>
#include <SD.h>
#include <cstdint>

#define NN_INPUT_SIZE 784
#define NN_HIDDEN_SIZE 297
#define NN_HIDDEN_SIZE_2 297
#define NN_OUTPUT_SIZE 250

int THETA = 2;

constexpr int kTensorArenaSize1 = 1024 * 10;
// Keep aligned to 16 bytes for CMSIS
alignas(16) uint8_t tensor_arena1[kTensorArenaSize1];

constexpr int kTensorArenaSize2 = 1024 * 10;
// Keep aligned to 16 bytes for CMSIS
alignas(16) uint8_t tensor_arena2[kTensorArenaSize1];

constexpr int kTensorArenaSize3 = 1024 * 10;
// Keep aligned to 16 bytes for CMSIS
alignas(16) uint8_t tensor_arena3[kTensorArenaSize1];


const tflite::Model* model1 = nullptr;
const tflite::Model* model2 = nullptr;
const tflite::Model* model3 = nullptr;
tflite::MicroInterpreter* interpreter1 = nullptr;
tflite::MicroInterpreter* interpreter2 = nullptr;
tflite::MicroInterpreter* interpreter3 = nullptr;
TfLiteTensor* input1 = nullptr;
TfLiteTensor* output1 = nullptr;
TfLiteTensor* input2 = nullptr;
TfLiteTensor* output2 = nullptr;
TfLiteTensor* input3 = nullptr;
TfLiteTensor* output3 = nullptr;


void normalize(float* input_data, int input_size){
  float sum = 0;
  for (int i = 0; i < input_size; i++) {
    sum += pow(input_data[i], 2);
  }
  for (int i = 0; i < input_size; i++) {
    input_data[i] = input_data[i] / (sqrt(sum)+ 0.0001);
  }
}

int label = 0;

void print_values(float* input_data, int input_size){
  for (int i = 0; i < input_size; i++) {
    Serial.printf("%f ", input_data[i]);
    if ((i+1) % 6 == 0) {
      Serial.println();
    }
  }
  Serial.println();
```

```
}


int read_example(File &file, float* pic) {
  // float pic[NN_INPUT_SIZE];
  if (!file) {
    Serial.println("File reading failed!");
    return;
  }

  char buffer[NN_INPUT_SIZE + 1];
  file.readBytes(buffer, NN_INPUT_SIZE+1); // Read a single label + example into buffer
  label = (int) (uint8_t) buffer[0];
  // Scale, quantize and copy data from buffer to input tensor
  for (int i = 0; i < NN_INPUT_SIZE; i++) {
    float scaled = ((float) (uint8_t) buffer[i+1]) / 255.0;
    pic[i] = scaled;
  }
  return label;
}


void encode_label(float *input_data, int label) {
  for (int i=0; i < 10; i++) {
    if (label == i ) {
      input_data[i] = 1.0;
    }
    else {
      input_data[i] = 0.0;
    }
  }
}

void setup() {
  // tflite::InitializeTarget();
  srand(millis());

  if (!SD.begin(BUILTIN_SDCARD)) {
    Serial.println("SD card initialization failed!");
    while(true);
  }
  tflite::InitializeTarget();

  static tflite::MicroMutableOpResolver<4> resolver1;
  static tflite::MicroMutableOpResolver<4> resolver2;
  static tflite::MicroMutableOpResolver<4> resolver3;

  resolver1.AddQuantize();
  resolver1.AddDequantize();
  resolver1.AddFullyConnected();
  resolver1.AddRelu();

  resolver2.AddQuantize();
  resolver2.AddDequantize();
  resolver2.AddFullyConnected();
  resolver2.AddRelu();

  resolver3.AddQuantize();
```

```
  resolver3.AddDequantize();
  resolver3.AddFullyConnected();
  resolver3.AddRelu();

  model1 = tflite::GetModel(layer_1);
  model2 = tflite::GetModel(layer_2);
  model3 = tflite::GetModel(layer_3);

  static tflite::MicroInterpreter static_interpreter1(model1, resolver1, tensor_arena1, kTensorArenaSize1);
  static tflite::MicroInterpreter static_interpreter2(model2, resolver2, tensor_arena2, kTensorArenaSize2);
  static tflite::MicroInterpreter static_interpreter3(model3, resolver3, tensor_arena3, kTensorArenaSize3);
  interpreter1 = &static_interpreter1;
  interpreter2 = &static_interpreter2;
  interpreter3 = &static_interpreter3;

  TfLiteStatus allocate_status1 = interpreter1->AllocateTensors();
  if (allocate_status1 != kTfLiteOk) {
    Serial.println("AllocateTensors() failed");
    return;
  }
  TfLiteStatus allocate_status2 = interpreter2->AllocateTensors();
  if (allocate_status2 != kTfLiteOk) {
    Serial.println("AllocateTensors() failed");
    return;
  }
  TfLiteStatus allocate_status3 = interpreter3->AllocateTensors();
  if (allocate_status3 != kTfLiteOk) {
    Serial.println("AllocateTensors() failed");
    return;
  }

  input1 = interpreter1->input(0);
  if (input1->type == kTfLiteInt8) {
        Serial.println("Model input type is kTfLiteInt8");
  } else if (input1->type == kTfLiteFloat32) {
    Serial.println("Model input type is kTfLiteFloat32");
  } else if (input1->type == kTfLiteUInt8) {
    Serial.println("Model input type is kTfLiteUInt8");
  } else if (input1->type == kTfLiteInt16) {
    Serial.println("Model input type is kTfLiteInt16");
  } else if (input1->type == kTfLiteInt32) {
    Serial.println("Model input type is kTfLiteInt32");
  } else {
    Serial.println("Model input type is unknown");
  }
  input2 = interpreter2->input(0);
  if (input2->type == kTfLiteInt8) {
        Serial.println("Model input type is kTfLiteInt8");
  } else if (input2->type == kTfLiteFloat32) {
    Serial.println("Model input type is kTfLiteFloat32");
  } else if (input2->type == kTfLiteUInt8) {
    Serial.println("Model input type is kTfLiteUInt8");
  } else if (input2->type == kTfLiteInt16) {
    Serial.println("Model input type is kTfLiteInt16");
  } else if (input2->type == kTfLiteInt32) {
    Serial.println("Model input type is kTfLiteInt32");
  } else {
    Serial.println("Model input type is unknown");
```

```
}
input3 = interpreter3->input(0);
if (input3->type == kTfLiteInt8) {
      Serial.println("Model input type is kTfLiteInt8");
} else if (input3->type == kTfLiteFloat32) {
  Serial.println("Model input type is kTfLiteFloat32");
} else if (input3->type == kTfLiteUInt8) {
  Serial.println("Model input type is kTfLiteUInt8");
} else if (input3->type == kTfLiteInt16) {
  Serial.println("Model input type is kTfLiteInt16");
} else if (input3->type == kTfLiteInt32) {
  Serial.println("Model input type is kTfLiteInt32");
} else {
  Serial.println("Model input type is unknown");
}

output1 = interpreter1->output(0);
if (output1->type == kTfLiteInt8) {
  Serial.println("Model output type is kTfLiteInt8");
} else if (output1->type == kTfLiteFloat32) {
  Serial.println("Model output type is kTfLiteFloat32");
} else if (output1->type == kTfLiteUInt8) {
  Serial.println("Model output type is kTfLiteUInt8");
} else if (output1->type == kTfLiteInt16) {
  Serial.println("Model output type is kTfLiteInt16");
} else if (output1->type == kTfLiteInt32) {
  Serial.println("Model output type is kTfLiteInt32");
} else {
  Serial.println("Model output type is unknown");
}
output2 = interpreter2->output(0);
if (output2->type == kTfLiteInt8) {
  Serial.println("Model output type is kTfLiteInt8");
} else if (output2->type == kTfLiteFloat32) {
  Serial.println("Model output type is kTfLiteFloat32");
} else if (output2->type == kTfLiteUInt8) {
  Serial.println("Model output type is kTfLiteUInt8");
} else if (output2->type == kTfLiteInt16) {
  Serial.println("Model output type is kTfLiteInt16");
} else if (output2->type == kTfLiteInt32) {
  Serial.println("Model output type is kTfLiteInt32");
} else {
  Serial.println("Model output type is unknown");
}
output3 = interpreter3->output(0);
if (output3->type == kTfLiteInt8) {
  Serial.println("Model output type is kTfLiteInt8");
} else if (output3->type == kTfLiteFloat32) {
  Serial.println("Model output type is kTfLiteFloat32");
} else if (output3->type == kTfLiteUInt8) {
  Serial.println("Model output type is kTfLiteUInt8");
} else if (output3->type == kTfLiteInt16) {
  Serial.println("Model output type is kTfLiteInt16");
} else if (output3->type == kTfLiteInt32) {
  Serial.println("Model output type is kTfLiteInt32");
} else {
  Serial.println("Model output type is unknown");
}
```

```

  Serial.printf("Input tensor dimension count: %d\n", input1->dims->size);
  Serial.printf("Input tensor dimensions: {");
  for (int i = 0; i < input1->dims->size; i++) {
    Serial.printf("%d", input1->dims->data[i]);
    if (i < input1->dims->size - 1) {
      Serial.printf(", ");
    }
  }
  Serial.printf("}\n");

  // print dimensions of output tensor
  Serial.printf("Output tensor dimension count: %d\n", output1->dims->size);
  Serial.printf("Output tensor dimensions: {");
  for (int i = 0; i < output1->dims->size; i++) {
    Serial.printf("%d", output1->dims->data[i]);
    if (i < output1->dims->size - 1) {
      Serial.printf(", ");
    }
  }
  Serial.printf("}\n");

  // print dimensions of input tensor
  Serial.printf("Input tensor dimension count: %d\n", input2->dims->size);
  Serial.printf("Input tensor dimensions: {");
  for (int i = 0; i < input2->dims->size; i++) {
    Serial.printf("%d", input2->dims->data[i]);
    if (i < input2->dims->size - 1) {
      Serial.printf(", ");
    }
  }
  Serial.printf("}\n");
  // output2
  // print dimensions of output tensor
  Serial.printf("Output tensor dimension count: %d\n", output2->dims->size);
  Serial.printf("Output tensor dimensions: {");
  for (int i = 0; i < output2->dims->size; i++) {
    Serial.printf("%d", output2->dims->data[i]);
    if (i < output2->dims->size - 1) {
      Serial.printf(", ");
    }
  }
  Serial.printf("}\n");

  // print dimensions of input tensor
  Serial.printf("Input tensor dimension count: %d\n", input3->dims->size);
  Serial.printf("Input tensor dimensions: {");
  for (int i = 0; i < input3->dims->size; i++) {
    Serial.printf("%d", input3->dims->data[i]);
    if (i < input3->dims->size - 1) {
      Serial.printf(", ");
    }
  }
  Serial.printf("}\n");
  // output3
  // print dimensions of output tensor
  Serial.printf("Output tensor dimension count: %d\n", output3->dims->size);
  Serial.printf("Output tensor dimensions: {");
  for (int i = 0; i < output3->dims->size; i++) {
```

```
    Serial.printf("%d", output3->dims->data[i]);
    if (i < output3->dims->size - 1) {
      Serial.printf(", ");
    }
  }
  Serial.printf("}\n");


  Serial.println("Model loaded successfully");
  delay(1000);

}

void print_pic(float* pic){
  for (int i = 0; i < NN_INPUT_SIZE; i++) {
    switch ((int) (pic[i] * 10))
    {
    case 0:
      Serial.printf(" ");
      break;
    case 1:
      Serial.printf(".");
      break;
    case 2:
      Serial.printf(":");
      break;
    case 3:
      Serial.printf("o");
      break;
    case 4:
      Serial.printf("O");
      break;
    case 5:
      Serial.printf("0");
      break;
    case 6:
      Serial.printf("&");
      break;
    case 7:
      Serial.printf("8");
      break;
    case 8:
      Serial.printf("%");
      break;
    case 9:
      Serial.printf("#");
      break;
    case 10:
      Serial.printf("@");
      break;

    default:
      Serial.printf(" ");
      break;
    }
    if ((i+1) % 28 == 0) {
      Serial.println();
    }
```

```
  }
}

void print_pic_num(float* pic){
  for (int i = 0; i < NN_INPUT_SIZE; i++) {
    Serial.printf("%f ", pic[i]);
    if ((i+1) % 28 == 0) {
      Serial.println();
    }
  }
}

float max(float* input_data, int input_size){
  // max value of input_data
  float max = 0;
  for (int i = 0; i < input_size; i++) {
    if (input_data[i] > max) {
      max = input_data[i];
    }
  }
  return max;
}

float min(float* input_data, int input_size){
  // min value of input_data
  float min = 0;
  for (int i = 0; i < input_size; i++) {
    if (input_data[i] < min) {
      min = input_data[i];
    }
  }
  return min;
}

float activation(float* input_data, int input_size){
  float sum = 0.0;
  for (int i=0; i < input_size; i++) {
    float y = input_data[i];
    sum += pow(y, 2);
  }
  return sum/input_size;
}

bool two_layers = true; // set to true if you want to use two layers
bool three_layered = false; // set to true if you want to use three layers
bool print_output = false;

void loop() {
  // Ask for mnist image input
  // tf tf_layer_2(static_interpreter1, NN_HIDDEN_SIZE, NN_OUTPUT_SIZE);
  // tf tf_layer_1(static_interpreter2, NN_INPUT_SIZE, NN_HIDDEN_SIZE);

  Serial.println("Starting inference on test set");
  File testFile = SD.open("fashion_mnist_test.bin");
  int correct = 0;
  int total = 0;
  float tot_time = 0;
  while (testFile.available()) {
```

```
    // Get a new entry from the file
    float pic[NN_INPUT_SIZE];
    label = read_example(testFile, pic);
    float begin = micros();
    if (print_output) {
      Serial.printf("Label: %d\n", label);
      Serial.println("Input");
      print_pic(pic);
      Serial.println("Input num");
      print_pic_num(pic);
      float mn= min(pic, NN_INPUT_SIZE);
      float mx= max(pic, NN_INPUT_SIZE);
      Serial.printf("Min: %f, Max: %f\n", mn, mx);
    }

    int predicted_label = 0;

    float max_goodness = -10;
    float goodness = -10;

    for (int i=0; i < 10; i++) {
      // float embed[NN_INPUT_SIZE] = {0.0};

      if (print_output) {
        Serial.printf("Encoding label: %d\n", i);
      }


      float sum = 0.0;


      // encode_label(embed, i);
      for (int j = 0; j < NN_INPUT_SIZE-1; j++) {
        if (j < 10){
          if (j == i) {
            input1->data.f[j] = 1.0;
          }
          else {
            input1->data.f[j] = 0.0;
          }
        }
        else {
          input1->data.f[j] = pic[j];
        }
      }


      if (print_output) {
        Serial.println("Encoded label");
        print_pic(input1->data.f);
        Serial.println("Encoded label num");
        print_pic_num(input1->data.f);
        float mn= min(input1->data.f, NN_INPUT_SIZE);
        float mx= max(input1->data.f, NN_INPUT_SIZE);
        Serial.printf("Min: %f, Max: %f\n", mn, mx);
      }


```

```
      interpreter1->Invoke();

      if (print_output) {
        Serial.println("First layer");
        print_values(output1->data.f, NN_HIDDEN_SIZE);
      }
      sum+=activation(output1->data.f, NN_HIDDEN_SIZE);

      if (two_layers) {

        for (int i = 0; i < NN_HIDDEN_SIZE; i++) {
          input2->data.f[i] = output1->data.f[i];
        }
        normalize(input2->data.f, NN_HIDDEN_SIZE);

        if (print_output) {
          Serial.println("Normalized layer");
          print_values(input2->data.f, NN_HIDDEN_SIZE);
        }

        interpreter2->Invoke();

        if (print_output) {
          Serial.println("Second layer");
          print_values(output2->data.f, NN_OUTPUT_SIZE);
        }

        sum += activation(output2->data.f, NN_HIDDEN_SIZE_2);

      if (three_layered){
        for (int i = 0; i < NN_HIDDEN_SIZE_2; i++) {
          input3->data.f[i] = output2->data.f[i];
        }
        normalize(input3->data.f, NN_HIDDEN_SIZE_2);
        if (print_output) {
          Serial.println("Normalized layer");
          print_values(input3->data.f, NN_HIDDEN_SIZE_2);
        }
        interpreter3->Invoke();
        if (print_output) {
          Serial.println("Third layer");
          print_values(output3->data.f, NN_OUTPUT_SIZE);
        }
        sum += activation(output3->data.f, NN_OUTPUT_SIZE);
      }
      }

      goodness = sum;

      if (print_output) {
        Serial.printf("Goodness: %f for label %d\n", goodness, i);
      }


      if (goodness >= max_goodness) {
        max_goodness = goodness;
        predicted_label = i;
      }
```

```
      if (print_output) {
        Serial.printf("Max goodness: %f for label %d\n", max_goodness, predicted_label);
      }


    }
    float end = micros();
    if (predicted_label == label) {
      correct ++;
    }
    if (print_output) {
      Serial.printf("Predicted label: %d, Actual label: %d\n", predicted_label, label);
      delay(50000);
    }


    total ++;
    tot_time += end - begin;

  Serial.printf("----------------------------------------\n");
  Serial.printf("SGD_FF_fashion_mnist_L297L297_BS32_E8_LR0.1_M0.0_0\n");
  Serial.printf("SGD_FF_fashion_mnist_L297L297_BS32_E8_LR0.1_M0.0_1\n");
  // Serial.printf("SGD_FF_fashion_mnist_L250L250L250_BS32_E6_LR0.1_M0.0_2\n");
  Serial.printf("Accuracy: %f\n", ((float) correct) / (float)total);
  Serial.printf("Average inference time (us): %f\n", tot_time / (float)total);

  }
}
```