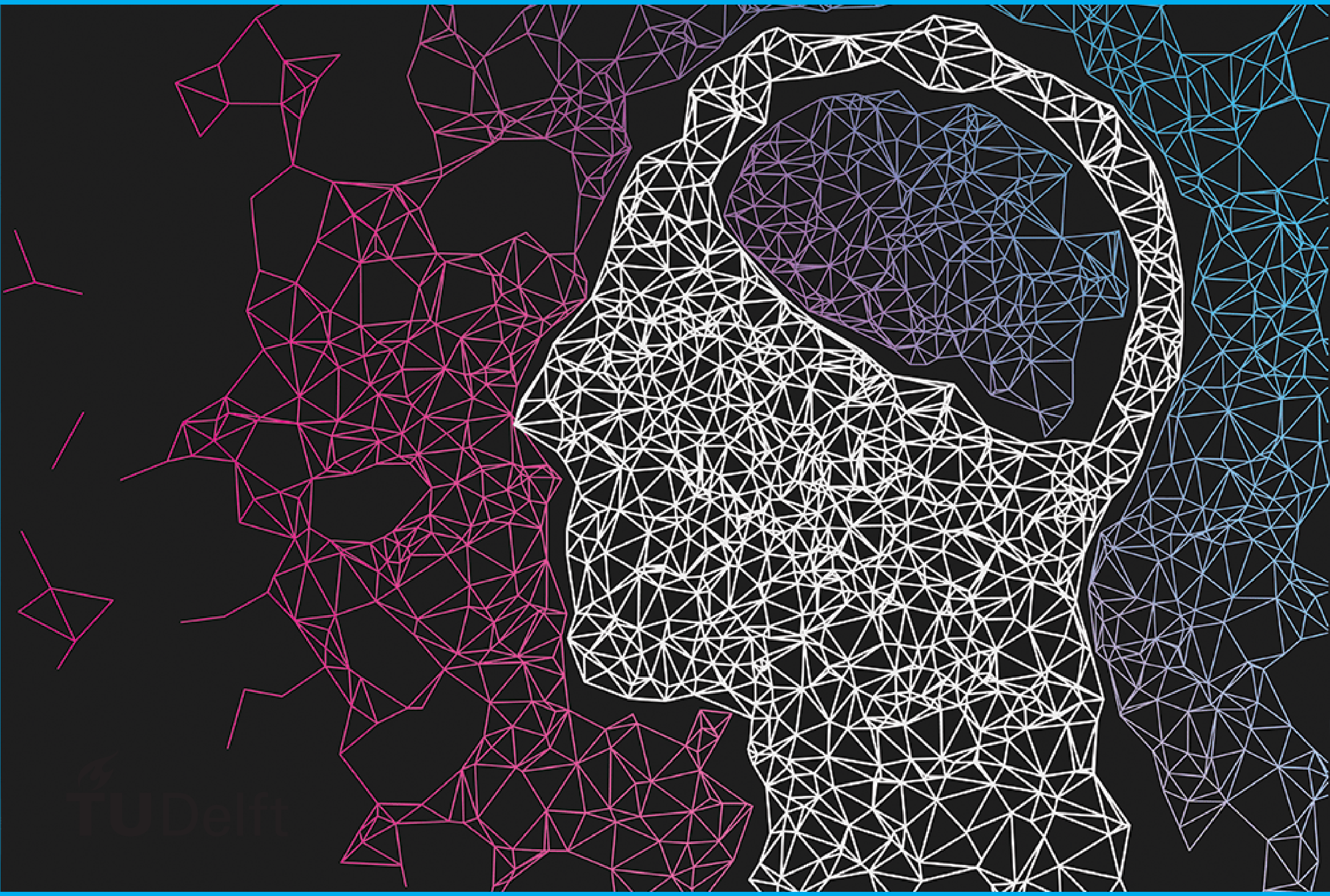
AI on Low-Cost Hardware

FPGA subgroup

Marijn Adriaanse & Li Ou Hu

Bachelor Thesis



AI on Low-Cost Hardware

FPGA subgroup

by

Marijn Adriaanse & Li Ou Hu

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Wednesday, June 21, 2023, at 14:30 AM.

Student number:	5346878 (Marijn Adriaanse)	
	5236541 (Li Ou Hu)	
Project duration:	April 24, 2023 – June 30, 2023	
Thesis committee:	dr. ir. J. H. G. Dauwels,	TU Delft, supervisor
	dr. C. Frenkel,	TU Delft, supervisor
	prof. dr. ir. M. Zeman,	TU Delft

TU Delft

Abstract

In the past decades, much progress has been made in the field of AI, and now many different algorithms exist that reach very high accuracies. Unfortunately, many of these algorithms are quite resource intensive, which makes them unavailable on low-cost devices.

The aim of this thesis is to explore algorithms and neural network techniques suitable for implementation on FPGAs. While FPGAs provide almost complete control over all aspects of design, allowing for the development of high-performance systems, they have not gained widespread popularity in neural network development due to their limited accessibility compared to computers and microcontrollers.

In the thesis, an inference-only 8-bit quantized neural net is designed, implemented and deployed on the Diligent Zedboard, and the performance is compared to similar networks on other devices. The thesis then focuses on two learning algorithms: Forward-Forward learning and Hebbian learning. It is shown how Forward-Forward can be seen as a way to apply Hebbian learning rules, and a simplified algorithm is proposed for use in a quantized system and implemented on an FPGA.

Although the performance of the network is quite low, reaching only 90.5% on the MNIST dataset and 74.2% on Fashion MNIST, the results are promising enough to give ground for further research and show that even very simplified versions of the Forward-Forward algorithm are capable of learning. Moreover, it demonstrates that the Forward-Forward algorithm is suitable for FPGA implementation.

Both implementations show that the processing speed of the FPGA implementations is much faster than that of similar network implementations on other devices.

Preface

After many hours of hard work, a lot of debugging, synthesizing, debugging (again), and the occasional screaming at the wall, we are proud to present our Bachelor thesis. Working with the FPGA proved to be more difficult and frustrating then anticipated, but in the end, we managed to get a satisfactory result.

We would like to thank our supervisors, dr. Charlotte Frenkel and dr.ir. Justin Dauwels, as well as prof. dr. ir. Frans Widdershoven and Mr. Yarib Nevarez, for their support throughout the project, and of course our other team members, Christiaan van den Berg, Jarl Brand, Manno Rom and Hong Jie Zheng.

This work marks the end of our time as Bachelor students. Though we certainly aren't leaving quite yet - we are both planning on doing the CESE Master here in Delft - we would like to take this opportunity to thank everyone at the university who made the past three years as great as they were.

We hope you enjoy reading this thesis and find it interesting.

Marijn Adriaanse & Li Ou Hu
Delft, June 2023

Contents

1 Introduction

In recent years, the rapid advancements in artificial intelligence (AI) and machine learning (ML) have led to significant breakthroughs in various domains. Neural network models, in particular, have emerged as powerful tools for solving complex tasks such as image classification, natural language processing, and speech recognition. However, traditional approaches to deploying these models often rely on cloud-based infrastructure or high-performance computing systems, which may be impractical or costly for low-cost embedded edge devices.

To address this limitation, a new paradigm known as on-device learning has gained traction. On-device learning refers to the ability of edge devices to train and update neural network models locally, without the need for constant connectivity to the cloud or reliance on powerful external servers. This approach offers several compelling advantages, including enhanced privacy, reduced latency, and improved energy efficiency.

Implementing neural network models with on-device learning capabilities in low-cost embedded edge devices presents a unique set of challenges. These devices typically have limited computational resources, restricted memory capacities, and power constraints. Furthermore, they often operate in resource-constrained environments with intermittent network connectivity, making traditional cloud-based training and inference methods impractical.

This thesis explores the state-of-the-art techniques and methodologies for implementing neural network models with on-device learning capabilities in Field Programmable Gate Arrays (FPGAs). FPGAs offer several advantages that make them a compelling choice for machine learning applications. FPGA design allows for full control over all operations and thereby also for more parallelism than microprocessors. This customizability makes extremely efficient implementations possible. Moreover, FPGA design is close to physical chip design, and many aspects of the FPGA design also transfer to chip design.

We will delve into the various architectural considerations, suitable algorithms, and model compression techniques that enable efficient on-device learning.

1.1. AI on low-cost hardware project

This thesis is the result of the FPGA subteam of the *AI on low-cost hardware* Bachelor Graduation Project. The goal of the project is to research different neural network techniques suitable for devices with limited resources.

The team, consisting of 6 students, is divided in three subteams of two students each. The software subteam deals only in the algorithms themselves, and does not attempt actual implementations. The second subteam implements algorithms on microcontrollers, and researches techniques for such platforms. Finally, there is the FPGA subteam, which has the objective of identifying techniques suitable for implementation on FPGAs.

The three subteams each have their own objectives, but work together for sharing techniques, algorithm and even complete models of networks. This also allows the team to compare similar systems on different hardware.

2 Background and Related works

This chapter covers several topics. It covers the basics of neural networks and the commonly used backpropagation algorithm, the MNIST and Fashion MNIST datasets, as well as some works related to neural networks on FPGAs.

2.1. Background on neural networks

Neural networks, also known as artificial neural networks (ANNs), are computational models inspired by the structure and functioning of biological neural networks in the human brain [1]. They are widely used in machine learning and artificial intelligence to solve complex problems by learning from data.

There are many types of artificial neural networks. This thesis only focuses on the most well-known type: a network in which the neurons are organized in layers, and each neuron is connected to every neuron in the next layer (the network is *fully connected*). Because there are no cyclical connections in the network, it is called a *feedforward* network.

The multilayer perceptron (MLP), shown in Figure 2.1, is a type of ANN that uses fully connected layers. This network type has three types of layers: the input layer (which is indicated in the figure as blue), a number of hidden layers (orange), and the output layer (purple).

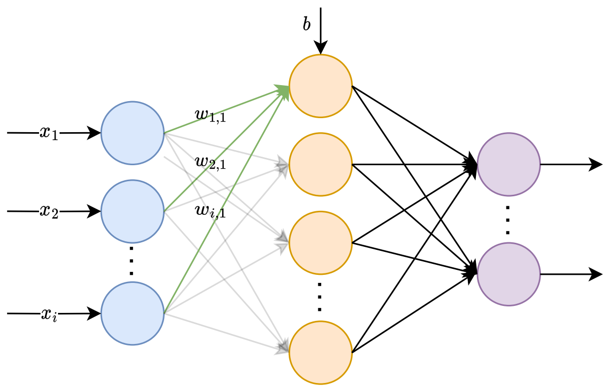


Figure 2.1: Multilayer perceptron with one hidden layer.

2.1.1. Common mathematics of artificial neurons

This section covers some of the most used mathematics behind neural networks [1]. Although there are networks that use different models, the mathematics discussed here apply to many different network types, including the networks used in this thesis.

The formulas here only describe the process of calculating the output of the neural network, which is called *inference*.

Generally, the output of a neuron is calculated by taking all the inputs, multiplying them by individual weights, taking the sum (these steps together constitute a *multiply-accumulate* (MAC) operation), adding some bias value, and applying a function called the *activation function*. For a neuron j with output y_j, M inputs x_i, weights w_{ij} connecting input i to neuron j, and bias b_j of neuron j, and activation function $f(\psi)$, the formula becomes:

$$y_j = f\left(\sum_{i=1}^{M} \left(w_{ij} \cdot x_i\right) + b_j\right) \tag{2.1}$$

The formulas can also be, and are usually, represented in matrix form, but this notation will be used, as it more clearly depicts the individual operations that will have to be implemented on the FPGA. A graphical representation of the formula can be seen in Figure 2.2.

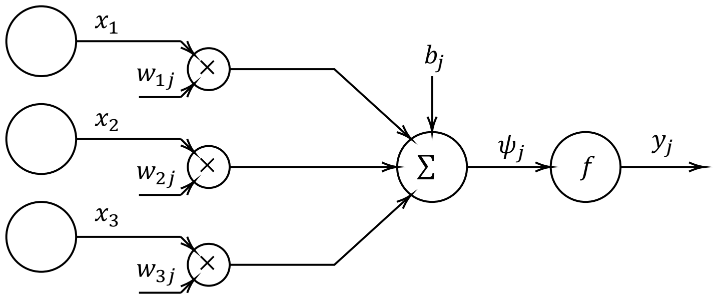


Figure 2.2: Graphical representation of the inference mathematics of a neural net.

For simplicity, the bias can be added as a weight to a input with a fixed value of 1 - the rest of this document assumes this configuration, and will not explicitly mention biases unless required.

$$y_j = f\left(\sum_{i=0}^{M} w_{ij} \cdot x_i\right),\ x_0 = 1,\ w_{0,j} = b_j \tag{2.2}$$

A large part of the behaviour of the network is determined by the activation function(s) used. There are several commonly used activation functions in ANNs, such as:

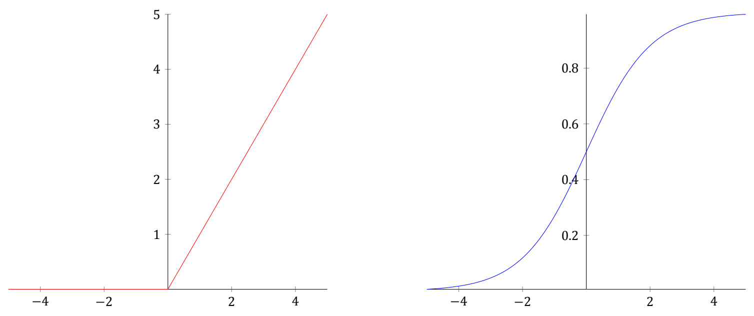
- **Rectified Linear Unit** (ReLU): ReLU activation sets negative values to zero and keeps positive values unchanged. It has the mathematical form: $f(x) = \max(0, x)$, as shown in Figure 2.3a ReLU is widely used due to its simplicity and ability to mitigate the vanishing gradient problem [2] .
- **Sigmoid** function (logistic function): This activation function smoothly maps the input to a range between 0 and 1 (Figure 2.3b). It has the mathematical form $f(x) = 1/(1 + e^{-x})$, and is closely related to the hyperbolic tangent, which maps values from -1 to 1.

In the project, the ReLU function was used, because its simplicity makes it better suited for implementation on FPGAs.

2.2. Backpropagation

Backpropagation is the most used learning algorithm in neural network training. The modern version of backpropagation was first published in 1970 [3], and has been used ever since.

The backpropagation algorithm works by calculating the output, comparing it to the desired output, and propagating the errors back through the network while updating weights along the way. This way, a network can be trained to give certain outputs for certain inputs.



(a) ReLU (b) Sigmoid function

Figure 2.3: Two neural network activation functions.

2.2.1. Mathematics

The backpropagation algorithm uses a measure of the difference between the actual and desired and actual outputs, called the *loss*, and updates weights based on the gradient of this loss with respect to the weights [4].

There are many different functions for calculating loss, such as the mean squared error (MSE). This determines the loss gradients of the last layer of the network.

The loss gradients are propagated back through the network to determine the gradient with respect to each individual weight. For outputs x_i of one layer, outputs y_j of the next layer, and respective pre-activation function sums ψ_i and ψ_j, connecting weights w_{ij}, activation function $f(\psi)$ and loss $\mathcal{L}$, the gradients of the loss are propagated through the network according to:

$$\frac{\partial \mathcal{L}}{\partial x_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \psi_j} \cdot w_{ij} = \sum_j \frac{\partial \mathcal{L}}{\partial y_j} \cdot f'(\psi_j) \cdot w_{ij} \tag{2.3}$$

and the gradients of the loss with respect to the weights is:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial \psi_j} \cdot x_i = \frac{\partial \mathcal{L}}{\partial y_j} \cdot f'(\psi_j) \cdot x_i \tag{2.4}$$

Here $\frac{\partial \mathcal{L}}{\partial \psi_j}$ is called the *local loss*. The propagation of the loss gradients through the network can be seen graphically in Figure 2.4. A more complete derivation can be found in section A.1.

The weights are changed according to:

$$w_{ij} := w_{ij} + \Delta w_{ij} \tag{2.5}$$

where the weight change Δw_{ij} depends on the gradient of the loss and the learning rate η:

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{L}}{\partial w_{ij}} \tag{2.6}$$

By applying the algorithm repeatedly, the loss is gradually decreased, as the outputs get closer to the desired outputs.

2.2.2. Implementation on FPGA

Because the backpropagation step requires previous outputs, all outputs need to be stored. This requires a lot of memory, especially for larger networks, making it less suitable for deployment on devices with limited resources.

In addition to its high memory usage, the backpropagation algorithm has more disadvantages when implementing it on an FPGA. The loss needs to be propagated back through the network, which comes down to having a secondary neural network that runs in the opposite direction, thus requiring more hardware resources for this implementation. With some care, most of the inference hardware can be reused, but this requires complicated rerouting of data. In any case, it greatly complicates the design.

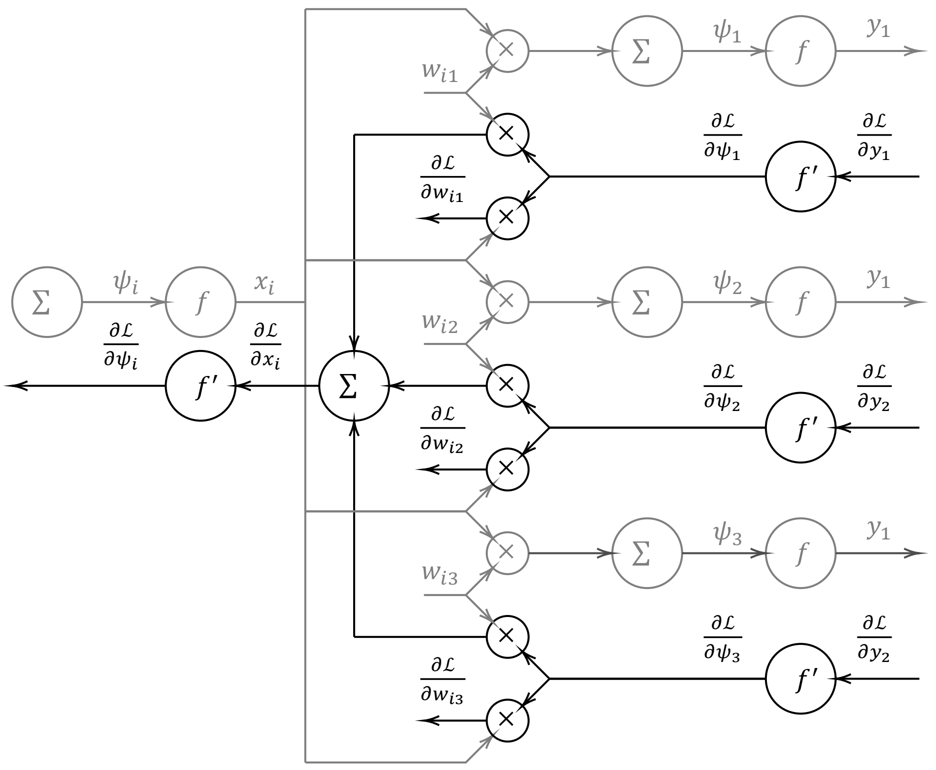

Σ
ψ_i
f
x_i
$\frac{\partial \mathcal{L}}{\partial \psi_i}$
f'
$\frac{\partial \mathcal{L}}{\partial x_i}$
w_{i1}
w_{i2}
w_{i3}
$\frac{\partial \mathcal{L}}{\partial w_{i1}}$
$\frac{\partial \mathcal{L}}{\partial w_{i2}}$
$\frac{\partial \mathcal{L}}{\partial w_{i3}}$
ψ_1
ψ_2
ψ_3
y_1
$\frac{\partial \mathcal{L}}{\partial \psi_1}$
$\frac{\partial \mathcal{L}}{\partial \psi_2}$
$\frac{\partial \mathcal{L}}{\partial \psi_3}$
$\frac{\partial \mathcal{L}}{\partial y_1}$
$\frac{\partial \mathcal{L}}{\partial y_2}$

Figure 2.4: Loss gradient propagation in a neural network using backpropagation.

2.3. The MNIST dataset

The MNIST dataset is a well-known collection of labeled grayscale images of hand-drawn digits [5]. The set is divided into 60,000 training images and 10,000 testing images. The images are 28 by 28 pixels, each having a value between 0 and 255.

The MNIST dataset is commonly used to test neural networks and other machine learning algorithms.

2.3.1. Fashion MNIST

The Fashion MNIST is a different dataset of the same format as MNIST, which depicts 10 types of clothing articles instead of digits [6]. It poses a more difficult classification problem than the MNIST dataset, because the different classes are less distinct, but it still has the exact same format as the MNIST dataset. This means it can be used to test models on a more difficult problem without changing the neural network.

2.4. Related works

The research on deploying machine learning workloads on edge devices is an ongoing field. With the increasing size of neural network models, the computational demands also grow proportionally. As a result, researchers have explored the feasibility of leveraging FPGAs for machine learning tasks. Numerous previous studies have effectively employed FPGA implementations for performing inference with MLP neural networks across diverse applications [7], [8].

As backpropagation is a commonly used algorithm for training neural networks, certain works have explored neural network training using traditional backpropagation techniques on FPGA, employing fixed-point arithmetic [9], [10].

Moreover, there have been additional investigations into the implementation of Hebbian learning on FPGA in separate studies [11]–[14]. These implementations either involve modifications to the neuron model, or use Spiking Neural Networks (SNNs).

In the design of the systems created, inspiration was taken from these works. However, it is worth noting that as of the writing of this paper, no existing works have been reported on the implementation of Forward-Forward learning on FPGA.

3 Programme of requirements

As the team was not provided with strict requirements, the team came up with their own requirements.

1. A baseline network implementation is created, that can perform inference of a pre-trained network.
2. An system is implemented that can perform on-device training using a method not requiring back-propagation.
3. The networks reach at least 90% accuracy on the MNIST dataset.
4. The networks reach at least 80% accuracy on the Fashion MNIST dataset.
5. The implementations are deployed on the Diligent Zedboard (provided for the project).

Considering the limited amount of time, and considerable development time of FPGA designs, only a baseline and single learning system were required. This system was not to use backpropagation, because of the problems discussed in subsection 2.2.2.

For testing the networks, the MNIST and Fashion MNIST datasets were chosen, as they are industry standards for testing neural networks and provide two difficulty levels that can be used with the same setup. Because they are commonly used datasets, the results can be compared to those of other methods. The accuracies can be considered easy to achieve using existing methods, but high enough to mark underdeveloped techniques as promising.

All other hard limitations of the design are imposed by the hardware provided for the project.

Besides the limitations imposed by the Diligent Zedboard and required accuracy levels, a lot of choices are left to the team. This results in a set of tradeoff requirements:

1. The accuracy of the models should be as high as possible.
2. The time required for inference should be minimal.
3. The time required for training should be minimal.
4. The design's footprint should be minimal.

4 FPGA platform specifications

For effective usage of the FPGA, its workings must be understood properly. This chapter discusses the Diligent Zedboard, which will be used for the project.

4.1. Hardware specifications

The Digilent ZedBoard, shown in Figure 4.1, is a development board equipped with the Xilinx Zynq-7000 System-on-Chip (SoC) [15]. This SoC contains a dual-core ARM Cortex-A9 processor tightly integrated with a Xilinx 7-series Field-Programmable Gate Array (FPGA). The ARM Cortex-A9 processor operates at a clock frequency of 650 MHz and has various periphery connected to it, such as an SD card interface and 512 MB of DDR3 memory.

In addition to the aforementioned features, the Digilent ZedBoard also offers a range of peripheral interfaces and connectivity options.

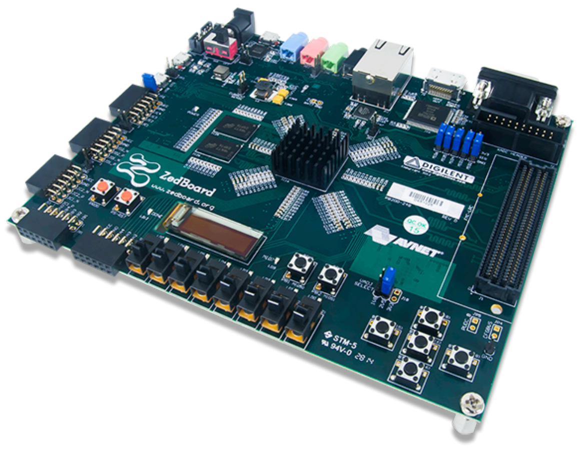


Figure 4.1: The Digilent Zedboard. Figure obtained from [15].

The FPGA section of the Zynq-7000 SoC can be configured at clock frequencies up to 250 MHz, providing flexibility in designing and optimizing the system's performance. The programmable logic (PL) in the FPGA is interconnected with the processing system (PS), which drives the clock.

4.2. FPGA components

The actual programmable logic consists of a large number of modules, that are configured to create the desired design. The FPGA has multiple types of these modules; the most important ones for this project are:

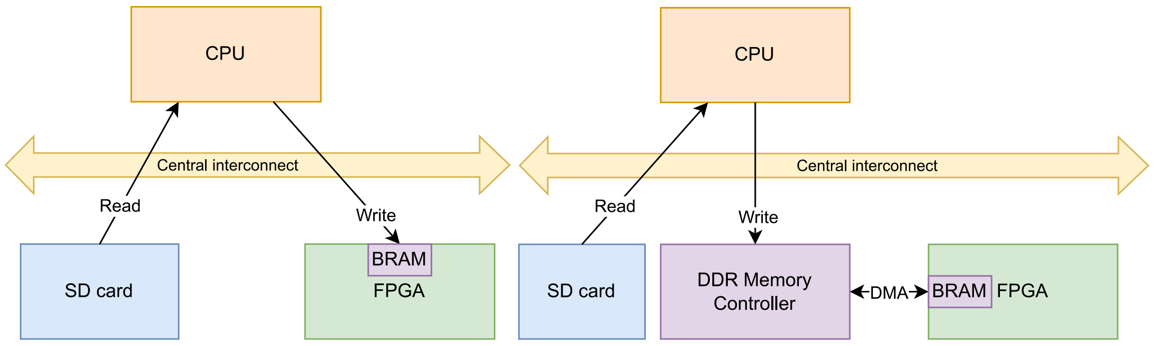
- 13300 slices: Slices these are the most generic module, that are used to create all sorts of logic, and even some memory. Together, the sliced provide 53200 Look-Up Tables (LUTs) and 106400 registers, which form the basis of all concurrent and sequential logic respectively.
- 140 Block RAM tiles: As the name suggests, these tiles are used as memory. Each tile holds $36Kb$, and can be used as two separate halves.
- 220 Digital Signal Processing (DSP) blocks: these module contain multiplication and accumulation functionalities.

Effective usage of the available components is essential for creating effective FPGA designs.

4.3. Data transfer

Although the neural network exists in the programmable logic, the data must be fed from the outside. In this project, the dataset will be placed on an SD card.

Multiple methods exist for transferring data from an SD card to an FPGA. The most straightforward approach involves instantiating GPIO lines. Another option is to employ an AXI-Lite interface, such as in Figure 4.2a, which provides memory-mapped registers for writing data from the PS (Processor System) side. For scenarios demanding increased bandwidth between the PS and PL, an AXI stream interface, shown in Figure Figure 4.2b, can be utilized. This interface incorporates Direct Memory Access (DMA) to facilitate data transfer of larger data blocks from external DDR3 memory to the PL. However, setting up a DMA transfer requires significantly more configuration in the software at the PS side. Thus, for this project, an AXI-Lite interface will be used.



(a) Data transfer using AXI-Lite or GPIO lines

(b) Data transfer using AXI-Stream DMA

Figure 4.2: Different system level architectures with PS and PL for transferring data to the FPGA.

5 Design considerations

Implementing systems on FPGAs comes with a number of design considerations that need to be made. The aim of this chapter is to provide a qualitative analysis of some general FPGA design aspects relevant in neural net design and of various techniques used in neural networks in light of FPGA implementation.

5.1. Limiting factors

There are a number of factors that limit available design choices. We have identified three that have a large impact on the entire design.

5.1.1. Size

An obvious limitation is the size of the FPGA, as the system must be synthesizable. The size of the system depends on many design aspects, such as the amount of parallelism and the quantization used, but also the arithmetic involved and the data paths required, and moreover, can be heavily dependent on small implementation details. Because nearly all facts of the design impact its size, size limitations result in extremely complex tradeoffs.

Due to the complexity of a design's size, it is difficult to accurately convey the available space of the FPGA.

5.1.2. Memory

The on-board memory of the FPGA is also a very important aspect to keep in mind. The memory requirements depend on the size of the neural network, the precision used, and the inference and learning algorithms.

If the required memory size exceeds the on-board storage size, data will have to be moved to an external memory. This limits the processing speed of the network to the bandwidth of the connection to the external memory, which is far lower than the achievable bandwidth on the FPGA itself.

The Zedboard has $4.9Kb$ of memory blocks embedded in the FPGA. For good performance, the network should not exceed this size, and some margins will be required to keep the design synthesizable.

5.1.3. Bandwidth

Even if the network fits completely on the FPGA, the network still needs to be supplied with data. Transferring data to and from the FPGA is limited to the speed of the communication method used. If the bandwidth is very low, the performance of the entire system will be low regardless of the speed of the network itself.

The AXI interface between the FPGA and microcontroller parts of the Zedboard can at most support $32bits$ per clock cycle. This is fast enough for input and output data, but not for the weights. Even DMA is only twice as fast. This once more stresses the importance of using the on-board memory cells, which allow a much larger bandwidth of thousands of bits per cycle.

5.2. Quantization

Quantized neural networks (QNNs) are neural network that utilize quantization techniques to represent and process network weights and activations using reduced precision. By reducing the number of bits used to represent these values, QNNs offer benefits such as a reduced memory footprint, faster computations, and improved energy efficiency.

In a traditional neural network, weights and activations are typically represented as floating-point values with a higher precision, such as 32-bit or 16-bit. The hardware required to perform computations on these numbers is large and slow. However, in QNNs, these values are quantized to a lower bit precision, commonly 8-bit or even lower. FPGA designs allow for complete customization of the quantization, and make using unusual data sizes such as 3 or 5 bits possible, that might be inefficient on computers and microcontrollers.

It is possible to convert a high-precision neural network to a quantized network. This is called *post-training quantization*. However, better results can be achieved if the training method takes the quantization into account: *quantization-aware training*. This usually still uses some higher precision values to update the network parameters.

It is much harder to do all training in a fully quantized environment. Low-precision quantization does not allow for small changes, as the total number of values is low. Furthermore, accurately representing, for example, gradients in few bits, is extremely hard [16].

On FPGAs, quantization is very important. Every bit saved can reduce memory a logic sizes, and small improvements on a synapse level can become vast when a network has many synapses.

The difficulties of quantized training have a large impact on both learning and inference in FPGA implementations. After all, if training requires high precision values, there is little benefit from using low precision values for inference, as (nearly) all hardware required for higher precision inference is already there because of the learning.

5.2.1. Floating point vs fixed point

Floating point numbers have the advantage of having a constant relative precision across a range of many orders of magnitude. This makes implementing neural networks easier, as the scaling is not that critical. However, it also requires more complicated hardware to perform even the simplest of operations.

The alternative is using fixed-point representations for numbers, which comes down to using integers representing multiples of small fractions (negative powers of 2). This comes at the cost of having to very carefully balance out the scaling in a network. Values that are too high are capped, and small values are lost to quantization. The optimal balance for the scaling depends not only on the network topology, but also the (distribution of) the input values and weights. This makes choosing proper values difficult.

[17] concludes that using floating point arithmetic is not worth the extra hardware, as in their MLP implementation fixed-point implementations are over 13 times smaller in area, and are over 12 times as fast.

5.2.2. Binary Neural Networks

Binary neural networks (BNNs) are an extreme type of quantized neural networks that use only two values (-1 and +1) for both the inputs, outputs and the weights of the network, and activations are done with the $\mathrm{sign}()$ function, resulting in a more efficient and hardware-friendly implementation.

Since each weight can be represented using only a single bit instead of multiple bits in traditional networks, the reduction in memory size can be up to $32\times$ [18], compared to `float32` weights. The binary nature of BNNs also leads to more efficient computations since binary operations are much simpler compared to both integer and floating-point operations.

Training binary neural networks poses unique challenges due to the discrete nature of binary weights, and traditional learning methods cannot be directly applied. Instead, various approximations and modifications have been proposed to train BNNs effectively. One common approach is the straight-through estimator [19], [20], which uses higher-precision weights for training. Implementing such learning on an FPGA would still require the infrastructure for higher-precision computations to be present, nullifying most of the benefits of the technique.

5.3. Implementing mathematical functions

Many neural network techniques work best with smooth functions. However, these tend to be computationally intensive. Although this problem also arises in conventional program-based implementations, it becomes more apparent on FPGAs.

Linear and logical operations are generally non-problematic (though multiplication can quickly increase the number of bits involved and may need clever scaling). These tend to make up the brunt of all operations in the network, as they occur for every two connected nodes.

The smooth functions involved are usually the activation function and loss. In many cases, smooth functions can be avoided by using functions like ReLU instead. If proper functions are absolutely required, a combination of look-up tables and interpolation can be used [21].

5.4. Implementing noise

Some neural network techniques require randomness [22], [23]. Random values are very difficult to create, but pseudo-random number generators can be relatively simple [24]. A single one of these units would be easy to implement, but if the noise is required at a synapse level, the synapses may need their own semi-independent noise sources. Simply using more noise sources would require those to be configured individually.

In reality, the algorithms may not actually need fully independent noise sources. A single noise source per input or neuron could prove to be sufficient, though it is likely that reusing a noise source would be more acceptable if the inputs and neurons involved are different. Even if they are reused, noise sources could be clocked, making the independence of the noise spread over multiple clock cycles. Finally, multiple bits could be cross-wired - potentially even being routed differently, to not only change the order of bits, but also spread the different bits across time.

Because it is quite an advanced technique, noise will not be used in the rest of this thesis.

5.5. Parallelism in the architecture

An important aspect of the architecture is parallelism. The amount of parallelism is proportional to the speed, but also (approximately) to the size of the design on the device.

There are various form of parallelism in neural networks [21]. At the lowest level, there is synapse level parallelism. This is probably the most important type of parallelism in any limited design, as the number of MAC operations (which occur at synapse level) are likely to be the limiting factor.

Next, there is neuron level parallelism, where neurons can exist in parallel. This would include having multiple activation functions in parallel. If the number of synapses is a bottleneck, neuron level parallelism might not be that useful, as a lower number of neurons with more synapses each could be used. Some network topologies might benefit from having more neurons with fewer synapses each, and offloading some parallelism of the synapses to the neurons could lower time constraints in the design.

Finally, there is layer level parallelism. This is most difficult to effectively implement, as different layers require each other's output, and generally have different sizes and thus timings. Having multiple layers in parallel could be an improvement if multiple full layers can be synthesized in parallel, or potentially if layers are so different they cannot use the same hardware.

Large amounts of parallelism might yield timing problems (i.e. because of large adders), requiring effective pipelining, and the amount of effective parallelism could be limited by the bandwidth of the memories involved.

5.6. Clock frequency

Though maybe not directly part of the design, the clock frequency has a large impact on the performance. The clock speed is directly proportional to the processing speed of the system (provided it does not need to wait on other systems with fixed frequencies). However, there are a few caveats: a high clock frequency comes with a proportional power increase, and the system needs to be designed for a high clock speed. Using a higher clock speed might require more pipelining to make sure all timing requirements can still be met.

6 Learning algorithms

Although backpropagation (see section 2.2) is the most used learning algorithm, many others exist. This chapter discusses two different learning algorithms: Forward-Forward and Hebbian learning, how they work and how well-suited they are for FPGA implementation. It also covers a few generic techniques, that can be applied to many learning algorithms.

6.1. Forward-Forward

Forward-forward learning [25] is a novel leaning algorithm that is similar to backpropagation, but trains layers individually. The network is fed both the input data and a label as input, and the activity of the layers is used to determine whether the network deems this label correct. Per layer, the weights are adjusted based on its inputs and outputs.

Because the forward-forward algorithm trains a single layer at a time, it does not require as much memory as the backpropagation algorithm. However, for classification problems, its single-value output requires performing inference for every class separately to obtain the most likely class.

To prevent a layer from simply learning to replicate the activity of the previous layer, the outputs are normalized between layers.

6.1.1. Mathematics

Like in backpropagation, the Forward-Forward algorithm uses gradient descent to update its weights:

$$w_{ij} := w_{ij} + \Delta w_{ij},\ \Delta w_{ij} = -\eta \frac{\partial \mathcal{L}}{\partial w_{ij}} \tag{6.1}$$

where $\mathcal{L}$ is the loss and η the learning rate. It can be shown that for the ReLU activation function and softplus ($\mathrm{softplus}(x) = ln(1 + e^x)$) loss function, these weight updates for correctly labeled inputs ($\Delta w_{ij,pos}$) and incorrectly labeled inputs ($\Delta w_{ij,neg}$) can be simplified to:

$$\begin{aligned} \Delta w_{ij,\ pos} &= \eta \mathrm{sig}(-\bar{\phi}) \cdot 2x_i y_j \\ \Delta w_{ij,\ neg} &= -\eta \mathrm{sig}(\bar{\phi}) \cdot 2x_i y_j \end{aligned} \tag{6.2}$$

where, for some offset $\bar{\theta}$, the average activity $\bar{\phi}$ of a layer of N neurons is obtained by taking

$$\bar{\phi} = \frac{1}{N} \sum_{j=1}^{N} (y_j^2 - \bar{\theta}) \tag{6.3}$$

Equation 6.2 will henceforth be written as:

$$\Delta w_{ij} = \pm \eta \mathrm{sig}(\mp \bar{\phi}) \cdot 2x_i y_j \tag{6.4}$$

The full derivation can be found in section A.2.

Between layers, the outputs are normalized. The normalized inputs $\hat{x}_i$ of the next layer become:

$$\hat{x}_i = \frac{x_i}{|\mathbf{x}|} = \frac{x_i}{\sqrt{\sum_{j=1}^{N} x_j^2}} \tag{6.5}$$

6.1.2. Implementation on FPGA

The Forward-Forward algorithm is far better suited for implementation on FPGAs than backpropagation. The weight changes requires only local information that is already present in a neuron, and information that is constant across a single layer. It does not require passing information back through the entire network, which simplifies the architecture and overcomes the problem of loss degradation found in backpropagation. Because layers are trained individually, it does not require information from other layers to be stored, save from the outputs of the previous layer.

The main drawback of Forward-Forward is that it requires output normalization. This means that each input of a layer depends on all other inputs. Because the outputs of a layer can only be scaled after all outputs have been calculated, and the scaling factor can vary wildly due to the very nature of the algorithm, proper scaling requires all outputs to be stored in a higher precision, that allows for this scaling without loss of precision.

Calculating the scaling factor itself is yet another problem - while accumulating the squared outputs of a layer is quite simple, calculating the inverse of the root of this sum is no trivial matter (as can be seen in Equation 6.5).

6.2. Hebbian learning

Hebbian learning is a nature-inspired learning mechanism, introduced by Donald Hebb in 1949 [26]. He stated that ”When one cell repeatedly assists in firing another, the axon of the first cell develops synaptic knobs (or enlarges them if they already exist) in contact with the soma of the second cell.” Informally, it is usually summarized as ”neurons that fire together, wire together”.

In its most basic form, the Hebbian learning rule defines the weight change Δw_{ij} for input x_i, output y_j, and learning rate η as: [27]

$$\Delta w_{ij} = \eta x_i y_j \tag{6.6}$$

6.2.1. Runaway weights

The basic form of the Hebbian learning rule has the problem that the weights are unbounded, and will grow to infinity if left unchecked. [27]

There are several solutions to this problem. The simplest way is to simply clip the weights. However, this can have detrimental effects on the learning mechanism, as other weights will continue growing, destroying the ratios between weights.

A different solution is the so called *Grossberg’s Instar Rule* [28], which limits the output through weight decay.:

$$\Delta w_{ij} = \eta y_j (x_i - w_{ij}) \tag{6.7}$$

Another very similar rule is *Oja’s rule*[29]:

$$\Delta w_{ij} = \eta y_j (x_i - y_j w_{ij}) \tag{6.8}$$

6.2.2. Other Hebbian learning rules

Assuming that both the inputs and outputs of the neurons are non-negative, it is easy to see that the Hebbian learning rule can only update weights upwards. As such, more similar rules can be created.

Most well-known is the so called anti-Hebbian learning rule, which adjusts weights downwards rather than upwards:

$$\Delta w_{ij} = -\eta x_i y_j \tag{6.9}$$

[30] proposes an additional set of learning rules for networks with binary outputs, that also cover cases where neurons don’t fire.

6.2.3. Supervised learning

In principle, Hebbian learning is unsupervised. The network learns the patterns it is fed, but does so on its own with no regard for desired output [31]. For a classification problem, such as the MNIST dataset, that is problematic, as the network must be trained on labeled inputs and must then predict a class.

One method to add supervision is to perform supervised learning only on the last layer of a network [27].

Another way to implement supervised learning is to add a label to the input, and have the network predict its correctness. Based on the result, (anti-)Hebbian and similar rules may be applied to the network. This technique is used in [30].

6.2.4. Implementation on FPGA

Hebbian learning has a number of beneficial properties regarding implementation on FPGAs. Because (standard) Hebbian learning only requires local knowledge, the architecture can remain simple. Additionally, updating the weights only requires simple linear operations, which should not be too difficult to quantize.

Supervised Hebbian learning does require some extra hardware. When applying different learning rules consistently across a network, these rules need to be distributed across all synapses, but this is still really simple. The most impactful part of such a system might be the way the learning rule is determined.

The implementation could be simplified by using fixed update sizes. This is for example possible in a network with binary (± 1) outputs, as presented in [30], as the weight changes also become binary updates.

6.3. General training techniques

There are some techniques that are not part of a single algorithm, but can be applied in many cases.

6.3.1. Learning rate decay

The learning rate is usually decreased throughout the training process. Although there is some debate on its effects [32], learning rate decay has shown to greatly improve training results.

Depending on the implementation, this might be difficult on FPGAs. Changing the learning rate itself is generally non-problematic, but very small weight updates require high precision weights. Quantization effects become more apparent as the weight changes get closer to the minimum weight size. At some point, a hard cutoff may be required.

6.3.2. Batches

In digital neural network implementations, networks often run both inference and learning on sets of multiple entries of a dataset, called *batches*, at once. This allows for more efficient computer resource utilisation, and allows for more stable gradient estimates [33].

Implementing such a mechanism on an FPGA would be costly. The network would have to store the stashed weight changes in addition to the actual weights, requiring more memory, and additional logic.

6.3.3. Momentum

Momentum is a technique that passes the weight changes through a low-pass filter. This can speed up learning and prevent the network from getting stuck on local minima [34].

As is also the case for batches, the use of momentum would require storing both the weight changes and weights themselves (though this is a drawback also found on computer implementations). However, slowly diminishing the value over time also requires much higher precision than simply accumulating weight changes, which in turn would take up considerably more additional memory. Furthermore, the multiplication steps require substantially more logic than a mere accumulator. These properties make momentum a very costly technique in FPGA design.

7 Experiments

Experiments were performed in two separate setups: one for 8-bit inference (section 7.1), and one for on-device single layer Forward-Forward and Hebbian based learning (section 7.2). Tests used the MNIST and Fashion MNIST datasets.

7.1. Inference of an 8-bit network

As a baseline implementation, an inference only setup is tested. The network uses 8-bit inputs, outputs and weights, making it comparable to similar implementations on other devices.

The network weights were trained on the Fasion MNIST dataset using backpropagation in TensorFlow and then quantized using the TensorFlow Lite optimizer [35]. The models were supplied by the software subteam.

Inference was performed on a 784-32-10 topology (784 inputs, 10 outputs, and one hidden layer of 32 neurons) and 784-32-32-10 topology (784 inputs, 10 outputs, and two hidden layers of 32 neurons). The topologies were chosen to allow comparison with the same model on other platforms.

7.1.1. Architecture

In the project, a flexible architecture was created. The neural net consists of the following parts:

- A **neuron block** consisting of a number of neurons.
- A **data register**, which holds both the input data and consequent layer outputs.
- A **weight register**, containing the weights of the network.
- A **controller**, which controls neuron behaviour, reading and writing.

In addition to the neural net itself, the design includes an AXI Lite interface for communicating with the CPU, which feeds it information, starts the inference and reads back the results. An overview of the system can be seen in Figure 7.1.

The neuron block handles MAC operations of N neurons for M inputs at once. All neurons are provided with the M shared inputs and their own M weights. For each block of N neurons in a layer, it accumulates the input-weight products of M inputs at once. Once all inputs have been accumulated, the outputs of the neurons are stored in the data register, and MAC operations of the next N neurons are started. Once all outputs in one layer have been calculated, the next layer is processed. Excess neurons can be disabled, and the first neuron can be configured to function as a bias by always outputting the maximum value. The value needs to be added to the inputs too.

The data register is divided into two parts: one is used to read inputs from, while the other is used to write outputs to. Between layers, this functionality is flipped, as the current layer's output become inputs for the next layer.

In the experiments, the values in the weight register were hard-coded. These could, for example, also be made non-static, allowing the CPU to change the network.

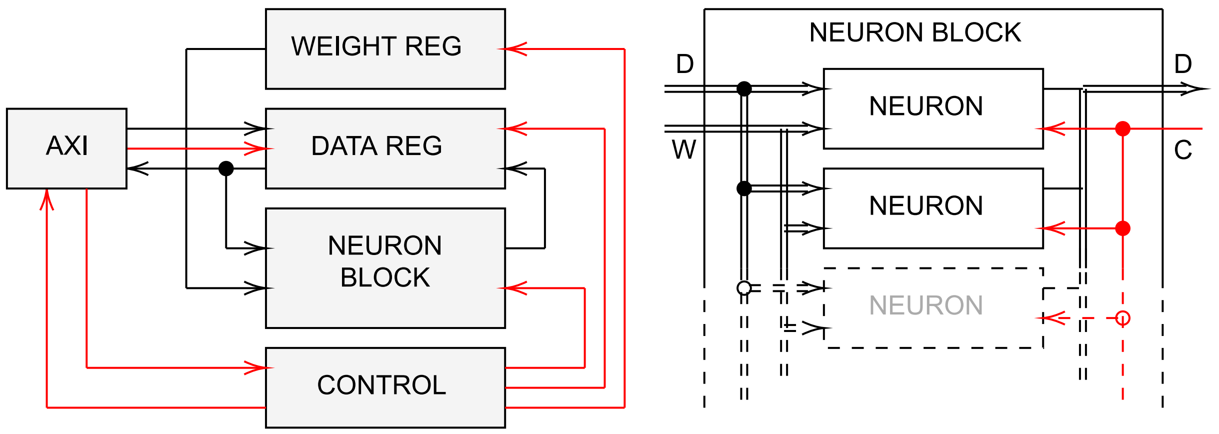


Figure 7.1: Architecture of the inference-only system. Control signals are marked in red.

The design size is mostly limited by the number of concurrent MAC operations, leaving the balance between N and M to be chosen. These can be chosen such that they best match the layer sizes - for example, if the network has a 128-32-16 neuron topology, it makes more sense to use 32 inputs and 16 outputs than to use 128 inputs and 4 outputs, because the second layer only contains 32 neurons. Of course, the number of neurons also impacts the design size, but still much less than the number of MAC operations. Finally, a large amount of parallelism in a single stage might require more pipelining. The experimental setup used 8 neurons with 8 inputs each.

The bias is created by adding a constant input, and adding a neuron to each neuron layer with a fixed output.

Because the inputs of the neuron block have a fixed size, the registers in the network use slightly more memory than required to represent the network. The weight register uses a number of bits equal to:

$$B_W = W \sum_{N,M} Q \left\lceil \frac{M+1}{Q} \right\rceil \cdot P \left\lceil \frac{N+1}{P} \right\rceil \tag{7.1}$$

where W is the number of bits used to represent weights, Q is the number of parallel inputs of the neurons, P is the number of neurons in the neuron block, M is the number of inputs of a layer and N the number of outputs. The $+1$'s in the equation are to account for the neurons used to provide the biases.

For the 784-32-10 topology, $B_W = 258560 bits \mathrel{\hat{=}} 32320 bytes$, and for the 784-32-32-10 topology, $B_W = 271360 bits \mathrel{\hat{=}} 33920 bytes$.

Similarly, the data register contains a number of bits equal to:

$$B_D = 2D \max(P,Q) \left\lceil \frac{N_{max}+1}{\max(P,Q)} \right\rceil \tag{7.2}$$

where D is the number of bits per input/output, N_{max} is the maximum number of inputs or outputs in a single layer in the network, and P and Q are the number of neurons in the neuron block and the number of parallel inputs of those neurons.

In this implementation, both halves of the register are large enough to store the results of any layer (or all inputs). This is suboptimal, and memory usage and could easily be reduced to:

$$B_D = D \left(\max(P,Q) \left\lceil \frac{N_{max,odd}+1}{\max(P,Q)} \right\rceil + \max(P,Q) \left\lceil \frac{N_{max,even}+1}{\max(P,Q)} \right\rceil \right) \tag{7.3}$$

where $N_{max,odd}$ and $N_{max,even}$ are the maximum number of neurons (counting the inputs as a layer of neurons) in odd and even layers.

For both topologies, $B_D = 12672 bits \mathrel{\hat{=}} 1584 bytes$, which could be reduced to $B_D = 6656 bits \mathrel{\hat{=}} 832 bits$.

Because it was designed to be flexible and reusable, the design is not fully optimized. Currently, MAC operations occur only once per 3 cycles, and thus better pipelining could speed up inference by almost a factor 3.

The AXI Lite interface is also sub-optimal. A higher data rate would be achievable using the full AXI protocol. Because the communication protocol was not part of the scope of the project, the AXI Lite interface was left in the design.

7.1.2. Quantization

Unfortunately, the subteam was unable to reproduce the scaling used by TFLite, and had to resort to quantizing the net. As pre-trained quantization is not a focus of the thesis, this was done in a very basic manner.

The floating point values of the weights were mapped from their full ranges to $[-127, 127]$, by scaling with the maximum absolute value divided by 127.49 and rounding the result. The biases were scaled with the same scaling factor, divided by 4. There is no particular reason for this scaling of the biases, other than the fact that it proved successful.

7.1.3. Scaling

After quantization, the matter of scaling the neuron outputs to 8-bit numbers need to be addressed. Because custom quantization was used, this also had to be done by the team.

The general procedure used for this is quite simple: run inference of all training entries up to a layer, look at the distribution of the outputs of the layer before scaling is applied, choose a suitable scaling factor which maps the values to the 8-bit range, and move to the next layer.

This process proved slightly more difficult due to the fact that the biases need their own scaling factors. The layer scaling was applied first, then the bias scaling was optimized, and finally the scaling factors were checked again.

To keep the scaling hardware-friendly, only bitshift operations were used, and distribution of the values was checked based on the number of values that require a certain number of bits. Only a single bias was tested for the entire networks.

For the 784-32-10 topology, the scaling chosen was 2^{-12} for the hidden layer and 2^{-7} for the final layer, and for the 784-32-32-10 topology, the first hidden layer was scaled by 2^{-11} (which caps a very small amount of values in the 12 bit range in favour of much more values in the 4 bit range) and the second hidden layer en output layer were scaled by 2^{81}. The scaling factor of 128 for the bias values worked well for both models. The distribution of unscaled output values per layer can be seen in Figure 7.2, and the complete values can be found in Table B.1.

This makes the final quantized and scaled activation function:

$$y_j = \mathrm{ReLU}\left(\left\lfloor S \cdot \left(\sum_{j=1}^{N}(w_{ij}x_i) + 128b_j\right)\right\rfloor\right) \tag{7.4}$$

for a layer of N neurons with scaling factor S as determined above.

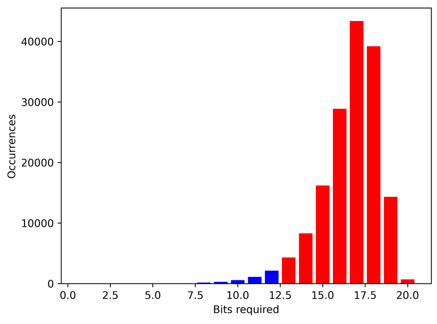
During the scaling process, it was found that the 784-32-10 topology was able to get 85.53% accuracy on the test data, and the 784-32-32-10 topology 85.67%.

7.1.4. Synthesis results

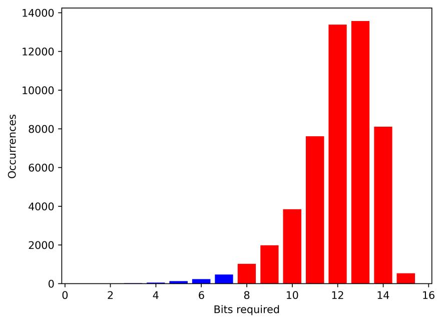
Designs for both topologies were successfully synthesized. The results can be seen in Table 7.1 and Table 7.2.

DSP blocks were used for the synapse MAC-operations, because these exactly incorporate the necessary operations. The results show that these are used for the 64 synapses that were modeled in parallel. The FPGA has enough DSPs to at least double this number to increase parallelism.

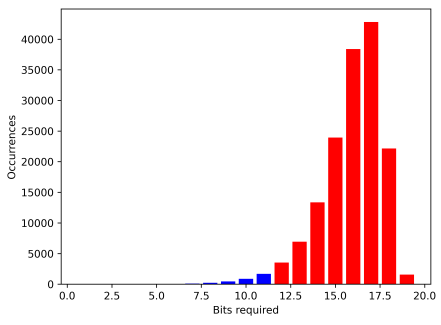
The data register uses 2 Block RAM tiles. Together these tiles can hold $72Kb$ of data, which is far more than the anticipated $12.4Kb$. Even if for both halves of the register space for 1024 values was reserved (rounding the actual number up to a power of two), this would only result in $8Kb$ of memory per half, and both halves should only require half a Block RAM tile. The reason why two full tiles are used is unclear.



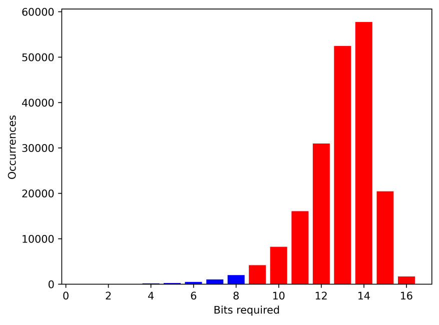
(a) 784-32-10, hidden layer



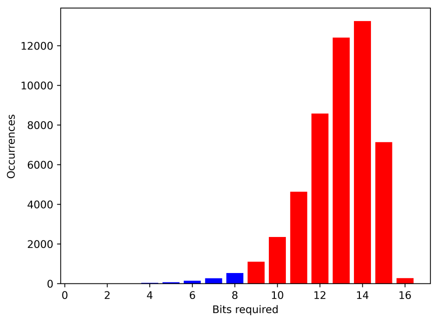
(b) 784-32-10, output layer



(c) 784-32-32-10, first hidden layer



(d) 784-32-32-10, second hidden layer



(e) 784-32-32-10, output layer

Figure 7.2: Distribution of pre-scaled outputs per layer of both network topologies. The most significant 8 bits are marked red.

The weight register uses 14.5 Block RAM tiles. This is also far more than expected - the weights should fit within 7.5 tiles. It seems that the amount of memory reserved is being round up to the next power of two, turning the $12672 bits$ required into 524288. Although this is unfortunate, it means that the design has more room for growth than would appear.

In both cases the controller uses very little resources, and the neuron block takes up the majority of the LUTs and registers in the design - this is fine, as it is where all of the parallelism in the design is located. Even then, only a small portion of the resources is used, because the relatively expensive MAC operations were offloaded to DSP block.

The design allowed running the system at up to $111.1 MHz$, which for FPGA standards is quite high.

Table 7.1: Synthesis results of the 784-32-10 topology.

Component	**Slice LUTs (53200)**	**Slice Reg. (106400)**	**B-RAM (140)**	**DSP (220)**
Complete design	1925 (3.6%)	886 (0.8%)	9.5 (6.8%)	64 (29.1%)
- Neural net	1379 (2.6%)	262 (0.2%)	9.5 (6.8%)	64 (29.1%)
- - Controller	83	36		
- - Neuron block	1233	226		64
- - Data reg.	64		2	
- - Weight reg.			7.5	
- Communication	197	193		
- Overhead	349	431		

Table 7.2: Synthesis results of the 782-32-32-10 topology.

Component	**Slice LUTs (53200)**	**Slice Reg. (106400)**	**B-RAM (140)**	**DSP (220)**
Complete design	1944 (3.7%)	887 (0.8%)	16.5 (11.8%)	64 (29.1%)
- Neural net	1397 (2.6%)	263 (0.2%)	16.5 (11.8%)	64 (29.1%)
- - Controller	92	37		
- - Neuron block	1250	226		64
- - Data reg.	64		2	
- - Weight reg.			14.5	
- Communication	197	193		
- Overhead	350	431		

The power usage was also estimated during synthesis. The results can be seen in Table 7.3. At just over $1.9W$, the system consumes quite a lot of power. However, most of this power is used by the processing systems, which include the onboard MCU. The other components only use $0.37 - 0.38W$ together. If the system were to be implemented on an FPGA without superfluous systems, the power consumption should be around this value.

7.1.5. Timing results

Of the synthesized design, timing measurements were taken. These can be seen in Table 7.4.

The inference times are as expected. The 784-32-10 topology takes 530 sets of 64 parallelized MAX operations, while the 784-32-32-10 topology takes 555. Because the design takes three cycles per set of MAC operations, and three additional cycles for writing outputs, this results in $13.83\mu s$ and $14.64\mu s$ respectively. The remaining $0.6\mu s$ is used for starting the network and detecting when it is finished. If the design were to be optimized to perform MAC operations every cycle, the neural network would take $4.57\mu s$ and $4.77\mu s$ respectively instead.

Table 7.3: Estimated power usage by the 8-bit inference implementations.

	784-32-10		784-32-32-10	
Component	**Power usage (W)**	**Percentage**	**Power usage (W)**	**Percentage**
Clocks	0.010	0.5	0.009	0.5
Signals	0.077	4.0	0.067	3.5
Logic	0.035	1.8	0.035	1.8
Block RAM	0.041	2.1	0.041	2.1
DSP	0.070	3.6	0.071	3.7
Processing system	1.558	80.3	1.558	80.7
Static power	0.150	7.7	0.149	7.7
Total	**1.941**	**100**	**1.930**	**100**

The communication delays are, though not unanticipated, quite large, taking up more than four times as much time as the actual inference. This is due to the inefficient communication protocol used. If a 32-bit number were to be transferred every clock cycle, the amount of time required would only be $1.8\mu s$, which - though it is significant - is at least much less than the inference time itself.

The current implementation can run inference on the training set in $3.9s$ and on the test set in $0.66s$.

Table 7.4: Timing measurements of the 8-bit inference implementation.

Step	**Time (μs) (784-32-10)**	**Time (μs) (784-32-32-10)**
Read SD	0.68	0.68
Write inputs	49.73	49.73
Inference	14.38	15.24
Read output	0.98	0.98
Total	**65.77**	**66.63**

7.1.6. Comparison

It is now possible to compare the performance of the system with implementations of the same network on other devices. For this, the results are compared to those of the TensorFlow Lite model, which the software and microcontroller subteams tested on a computer and microcontroller. The results can be found in Table 7.5.

The 784-32-10 topology was able to get 85.53% accuracy on the test data, and the 784-32-32-10 topology 85.67%. This is up to a couple percent lower than the models quantized using TensorFlow Lite, which was to be expected due to the simplistic and unoptimized quantization.

The comparison shows that the FPGA implementation, excluding data transfers to and from the network, is more than twice as fast as the TensorFlow model on a PC, and more than four times as fast as the same model on a microcontroller. Moreover, because the inference time is very short, the network uses $39 - 42\%$ less energy per inference sample.

These results were achieved at quite a high clock frequency, but the model can still be improved. The model is currently not as pipelined as it could be, and the inference time on the FPGA itself could be reduced by a factor 3. This would both lower the inference time per sample greatly (though the communication delays in the reported inference time would remain the same) and reduce the energy costs per sample. Moreover, using a more efficient communication protocol could greatly reduce the time required to write inputs and read outputs, and the FPGA implementation would be much faster than the other implementations when including the data transfers.

It should of course be noted that the TensorFlow model is slightly more complex than the FPGA implementation. It is however believed that the computations in the TensorFlow model could be adequately implemented on the FPGA without requiring a lower clock frequency, if effective pipelining is carefully observed.

Table 7.5: Comparison between different implementations of an 8-bit network on different devices.

Topology	784-32-10			784-32-32-10		
Implementation	TFLite		VHDL	TFLite		VHDL
Platform	PC[1]	MCU[2]	FPGA	PC[1]	MCU[2]	FPGA
Accuracy (%)	86.33		85.53	87.36		85.67
Inference time/sample (μs)[3]	34.62	70.19	14.38	37.47	75.20	15.24
Inference energy/sample (μJ)		46	28[4]		50	29[4]

7.2. On-device single layer learning

The second set of experiments includes on-device learning. The algorithm used is custom, and was based on both Forward-Forward learning, though in practice it is also a form of supervised Hebbian learning.

Based on results of tentative testing, the decision was made to use about 100 neurons. The actual number used is 104, as multiples of 8 are preferred in the architecture. Because the algorithm requires the classification labels to be specified as input, this means a 794-104 topology was used (excluding a single constant input to act as a bias). Larger networks may get a slightly higher accuracy, would have been much slower to simulate during development.

7.2.1. Algorithm

The algorithm was based on the Forward-Forward algorithm implementation of the software subteam. Because of the need for output normalization between layers, and promising results from the software and microcontroller subteams when using only a single layer, a single layer setup was chosen.

Taking Equation 6.4 from Forward-Forward learning, we can see that the formula can be divided in a layer and synaptic term:

$$\Delta w_{ij} = \pm[2\eta \mathrm{sig}(\mp\bar{\phi})] \cdot [x_i y_j] \tag{7.5}$$

As the first term is constant across the entire layer, and the second is the Hebbian learning rule, it becomes apparent that this method is effectively applying Hebbian and anti-Hebbian learning, with a variable learning rate scaled by the sigmoid function. Comparing Equation 7.5 to Equation 6.6 and Equation 6.9, we see that:

$$\eta_{Hebbian} = \eta_{FF} \cdot 2\mathrm{sig}(\pm\bar{\phi}) \tag{7.6}$$

One important change was made to the algorithm: the weight change was changed to fixed values of ± 1 or 0. This vastly simplifies the weight update procedure and solves scaling problems, but it does change the algorithm significantly. Simulations show that the algorithm can still function.

The initial thought for converting the arbitrarily valued weight change to these fixed values is be to take the sign function of the original weight change:

$$\Delta w'_{ij} = \mathrm{sign}(\Delta w_{ij}) = \pm\mathrm{sign}(\mathrm{sig}(\mp\bar{\phi})) \cdot \mathrm{sign}(x_i) \cdot \mathrm{sign}(y_j) \tag{7.7}$$

Technically the sigmoid function will never reach 0 and always have a sign of $+1$, but it does vanish towards minus infinity and needs an approximation that actually reaches 0. The softplus function used for the loss, which the sigmoid function was the derivative of, is essentially a smooth version of the ReLU function - if we were to take its derivative instead, the sigmoid would turn into a step function. At first glance, this might seem a suitable approximation. However, with the softplus loss function, still a considerable amount of learning is done at $\phi = 0$. Simulations indicate that it might be better to choose a shifted step function:

$$\mathrm{sign}(\mathrm{sig}(\mp\bar{\phi})) \rightarrow \mathrm{step}(\bar{\phi} + \bar{\phi}_{sigmoid}) \tag{7.8}$$

with, as a tentative value, $\bar{\phi}_{sigmoid} \approx 1$. Finally we find:

$$\Delta w'_{ij} = \pm\mathrm{step}(\bar{\phi} + \bar{\phi}_{sigmoid}) \cdot \begin{cases} 1, & \text{if } x_i > 0 \ \& \ y_j > 0 \\ 0, & \text{otherwise} \end{cases} \tag{7.9}$$

[1]Intel(R) Xeon(R) CPU, $2.00 - 2.30 MHz$, 1 core, 2 threads, $38.5 - 55MB$ L3 cache.
[2]Teensy 4.1 running at 600MHz.
[3]Not including data transers.
[4]Value obtained by multiplying full estimated power by inference time excluding SD card reads.

Using this formula, the sign is purely determined by whether the input label is correct, synapses are updated if both the pre-synaptic and post-synaptic neurons have positive output, and learning can be omitted altogether if $\mp\bar{\phi} < -\bar{\phi}_{sigmoid}$. This makes the algorithm easily implementable and hardware efficient.

The network is trained by training on a correctly labeled entry in the dataset, and then training it on the same entry with a random incorrect label. This process is repeated for all entries.

7.2.2. Architecture

The architecture of the learning implementation is similar to that described in subsection 7.1.1. As the algorithms and topology have changed, there are a few modifications to the design:

- The neuron block was fitted with an accumulator, which adds the squared neuron outputs to determine the activity of the layer. The activity is now the only output of the system.
- Neuron outputs are no longer stored in the data register, as there are no next layers requiring this information. Instead, this register is now only used for the input data.
- An output register was added; this register stores whether the output is non-zero for every neuron in the topology, to speed up learning.
- Squared and non-zero outputs were added to the neurons for the layer activity and weight updates.
- The weight update logic was added to the synapses.
- The controller was re-written to suit the new architecture.

The new architecture can be seen in Figure 7.3.

As the learning algorithm requires the outputs of the neurons to change the weights, and calculating the output requires iterating over all inputs and weights, updating the weights takes at least 2 passes. To prevent the system from having to do 3 passes in total (one to determine whether learning needs to applied, one to recalculate the outputs, and one to update the weights), the outputs are stored in the output register and fed back into the neurons for the learning pass. Because the weight updates only depend on whether the output was non-zero, only this information is stored.

The system was configured with 8 neurons with 16 parallel inputs each.

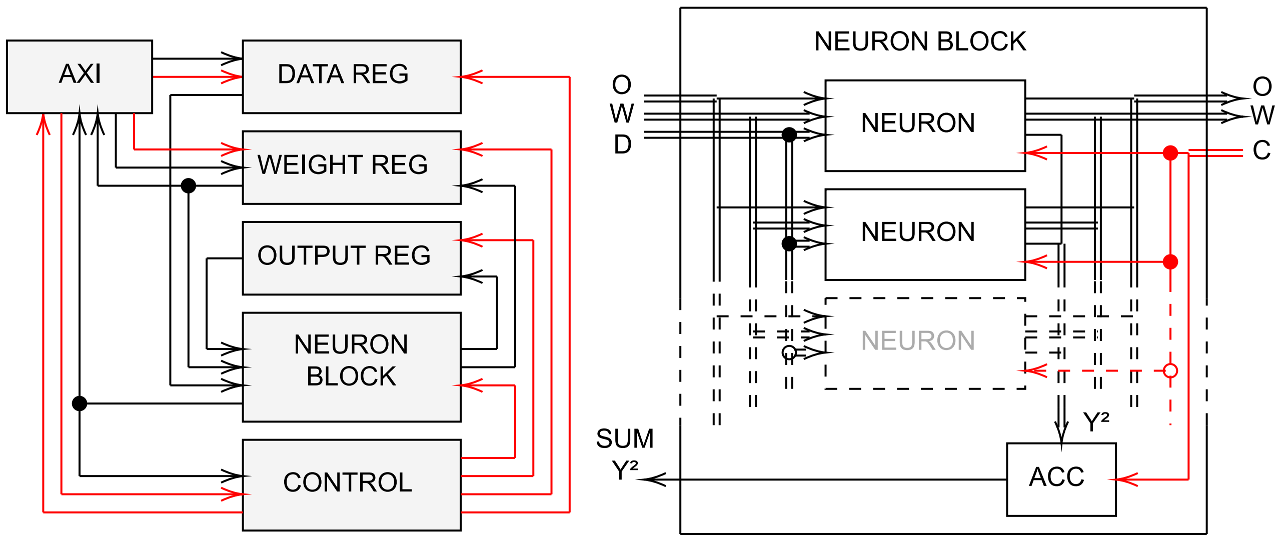


Figure 7.3: Architecture of the on-device learning system. Control signals are marked in red.

Because network has changed, so have the sizes of the registers. The weight register now uses a number of bits equal to:

$$B_W = W \cdot Q \left\lceil \frac{M+1}{Q} \right\rceil \cdot N \tag{7.10}$$

as there is now only one layer with M inputs (including the 10 inputs for the class labels) and N outputs. The design was simplified to allow only multiples of P as the number of neurons.

For the chosen 794-104 topology, $B_W = 1331200 bits \mathrel{\hat{=}} 166400 bytes$.

Similarly, now that the data register only hold the inputs, it contains a number of bits equal to:

$$B_D = D \cdot Q \left\lceil \frac{M+1}{Q} \right\rceil \tag{7.11}$$

where, again M is the number of inputs including the class labels.

For the chosen topology, $B_D = 6400 bits \mathrel{\hat{=}} 800 bytes$.

Finally, the output register uses single bit per neuron:

$$B_O = N \tag{7.12}$$

which is $104 bits \mathrel{\hat{=}} 13 bytes$ for the 794-104 topology.

7.2.3. Quantization and scaling

The quantization was done using 8-bit inputs and outputs and 16-bit weights. The total activation, which is used as the output of the network, is 32 bits.

Using 8 bits for the input makes sense, as this is the format of the data in the MNIST dataset. The 8-bit output was chosen rather arbitrarily to have a similar precision. This results in 16-bit squared outputs.

By adding no more than a couple hundred of these 16-bit numbers, it would be impossible to get even remotely close to 32 bits. This number was chosen to facilitate interfacing with the CPU part of the system, and the extra bits should have minimal impact on the area used.

The 16-bit weights were chosen for similar reasons. We were unable to get the learning working with only 8-bit weights, likely due to the large relative size of the weight updates, and 16 bits are the next divisor of 32 bits. Through simulations, it was found that the weights actually only require 11 bits before experiencing an accuracy drop.

Some scaling had to be done to keep the values in range:

$$y_j = \min\left(\left\lfloor \frac{\mathrm{ReLU}(\psi_j)}{2^{12}} \right\rfloor, 255\right)$$

$$\bar{\phi}_{sigmoid} = 40$$

$$\bar{\theta} = 80$$

These values were found empirically, and have little meaning beyond the interpretation of the inputs and weights as fixed point numbers. What is important, is that $\bar{\theta} = 2\bar{\phi}_{sigmoid}$: they have the same relative value as was the case for $\bar{\theta} = 2$ and $\bar{\phi}_{sigmoid} = 1$.

The weights were initialised with random values on $[-160, 160]$.

Though these hyperparameters do affect the accuracy to some extent, the network does not seem to be very sensitive to them. The time limitations did not allow for an extensive hyperparameter search, as the lack of batches makes computer implementations slow, and synthesis takes even more time. The only way to do a proper search would be to do so on-device, which would require more development beyond the time limitations.

7.2.4. Synthesis results

The design was successfully synthesized. The resource usage can be seen in Table 7.6. Due to timing issues, the clock frequency had to lowered to $71.4 MHz$.

The control unit has increased drastically in size. This is likely due to the addition of the comparison logic of the summed squared outputs and extra pipelining.

The neuron block uses significantly more resources than in the inference-only system. This was to be expected, as the number of neurons was doubled, and weight modifications, squared outputs and squared output accumulation were added. As with the inference-only design, the DSP blocks were used for MAC-operations. The number of DSPs used is as expected: 16 inputs times 8 neurons results in 128 synapses.

The data register was synthesized using LUTs, registers and multiplexers, while the weight register still uses the Block RAM tiles. This change in the synthesis of the data register is likely due to the fact that the data register has different input and output sizes in this design (the input size is 32 bits to allow for easier writing from the communication modules, while the output size is $16 \times 8bits$).

Again, the number of Block RAM tiles is much larger than expected. The weights only take up 1331200 bits, which is 25.8% of the available $4.9Mb$. Like in the inference-only design, the amount of memory synthesized is rounded up to a power of two, resulting in the usage of 57 tiles.As there are only 140 Block RAM tiles, the maximum number of neurons in the design in its current state is 256, as this would result in the use of 114 of the Block RAM tiles.

The output register is extremely small, storing only 100 bits, and thus uses barely any resources.

The communication modules use quite a lot of LUTs and registers, which are mainly in the input buffers. The cause of this is unknown, but as the thesis focuses on the network itself, this was not further investigated.

During synthesis, the power usage was also estimated. The results can be seen in Table 7.7. As was also the case, the processing systems use most of the power, not the actual FPGA.

The power usage is quite similar to that of the 8-bit system. This might seem odd at first, since that design is definitely smaller in size, but for the 8-bit inference, a higher clock speed was used, resulting in higher relative power consumption.

Table 7.6: Synthesis results of the learning system.

	Slice LUTs (53200)	**Slice Reg. (106400)**	**F7 MU (26600)**	**F8 MU (13300)**	**B-RAM (140)**	**DSP (220)**
Complete design	18178 (34.2%)	13871 (13.0%)	768 (2.9%)	384 (2.9%)	57 (40.7%)	128 (58.2%)
- Neural net	11449 (21.5%)	10980 (10.3%)	768 (2.9%)	384 (2.9%)	57 (40.7%)	128 (58.2%)
- - Controller	330	82				128
- - Neuron block	9561	4362				128
- - Data reg.	1552	6528	768	384		
- - Weight reg.					57	
- - Output reg.	6	8				
- Communication	6355	2432				
- Overhead	374	459				

Table 7.7: Estimated power usage of the system.

Component	**Power usage (W)**	**Percentage**
Clocks	0.037	1.9
Signals	0.108	5.6
Logic	0.083	4.3
Block RAM	0.092	4.8
DSP	0.037	1.9
Processing system	1.557	81.3
Static power	0.158	8.3
Total	**1.915**	**100**

7.2.5. Accuracy results

10 different runs of the algorithm, implemented in Python, were performed on the MNIST dataset and the obtained accuracies (on the test data) and losses after every epoch were recorded. The results of the first five runs, as well as the averages, can be seen in Figure 7.4 and Figure 7.5. The full results can be found in Table C.1 and Table C.4. The loss was calculated using $\mathrm{softplus}(p\bar{h}i/\bar{\phi}_sigmoid)$, based on the notion that a value of $\bar{\phi}$ equal to $\bar{\phi}_{sigmoid}$ should represent a value of about 1.

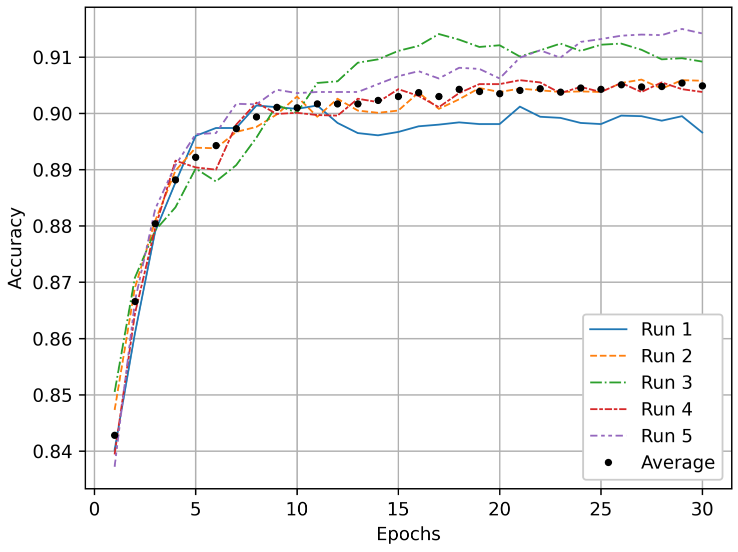


Figure 7.4: Accuracies of the learning model when trained on the MNIST dataset.

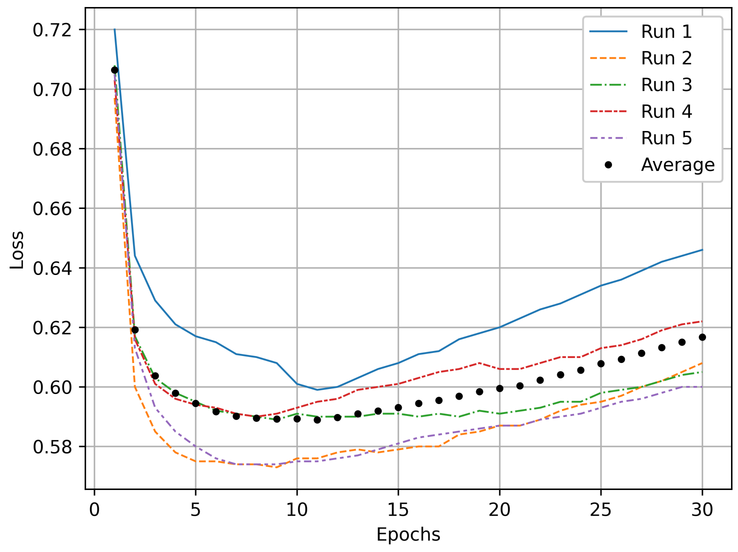


Figure 7.5: Losses of the learning model when trained on the MNIST dataset.

The network clearly learns quite rapidly. After 10 epochs, all runs were very close to 90% accuracy, with an average accuracy of 90.1%. After that, the values tend to fluctuate a bit, with one run managing to reach 91.4%. In the remaining epochs, the average accuracy every so slowly increases, reaching a maximum at epoch 29 of 90.5%. This demonstrates the learning algorithm is capable of reaching 90%, though barely.

What is very interesting is that the loss reaches a minimum at somewhere between epoch 6 and 12, and actually loss goes *up* afterwards, even though (generally) the accuracy is not decreasing. The average loss is minimal in epoch 11, which more or less coincides with the start of random fluctuations in the accuracy.

This behaviour indicates two things. First of all, the softplus function is no longer an accurate function for estimating the accuracy of the model. This is not that strange - after all, the model was changed in a way that completely removes the softplus function from any of the mathematics involved. Secondly, the training seems to happen in two phases. During the first phase, the softplus loss goes down, and the model reaches an accuracy close to its maximum. During the second phase, the loss goes back up, and the model barely improves. This seems to generally still have a positive effect on the accuracy, but it is very minor. The test size of 10 runs is simply not large enough to conclude decisively.

Additionally, 6 different runs of the algorithm on the Fashion MNIST dataset were recorded. The results of the first five runs, as well as the averages, can be seen in Figure 7.6 and Figure 7.7. The full results can be found in Table C.2 and Table C.5.

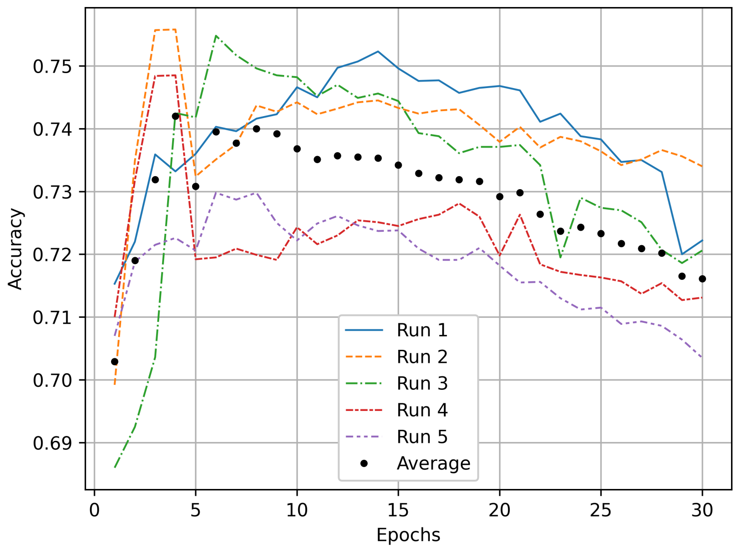


Figure 7.6: Accuracies of the learning model when trained on the Fashion MNIST dataset.

Immediately, it is clear that the algorithm does not perform nearly as well as it did on the MNIST dataset, reaching no more than 75.6% in the best case and 74.2% on average. Unfortunately, this means it did not make the requirement of getting 80% accuracy on Fashion MNIST.

Some interesting things are happening here. Two of the runs reach their maximum accuracy after 3 epochs, to then go down steeply after 5, and in total 5 runs reach their maximum after no more than 6 epochs. Run 1 is the only one that much more gradually increases, reaching its maximum accuracy after 14 epochs. And unlike the runs on the MNIST dataset, the accuracy then clearly goes down, reaching an average of 71.6% after 30 epochs. This is not simply a case of overfitting, as during testing it was noticed that the accuracy on training data also decreases.

For all runs, the loss reaches a minimum after epoch 2, after which it rapidly increases. In half the cases, the loss seems to more or less stagnate around epoch 15. These runs also perform considerably better in terms of accuracy after those 15 epochs.

Overall, the behaviour of the algorithm on the Fashion MNIST set is slightly different, having large drops in accuracy, reaching the minimum loss in only two epochs, and experiencing an overall drop in average accuracy when the network is trained for many epochs.

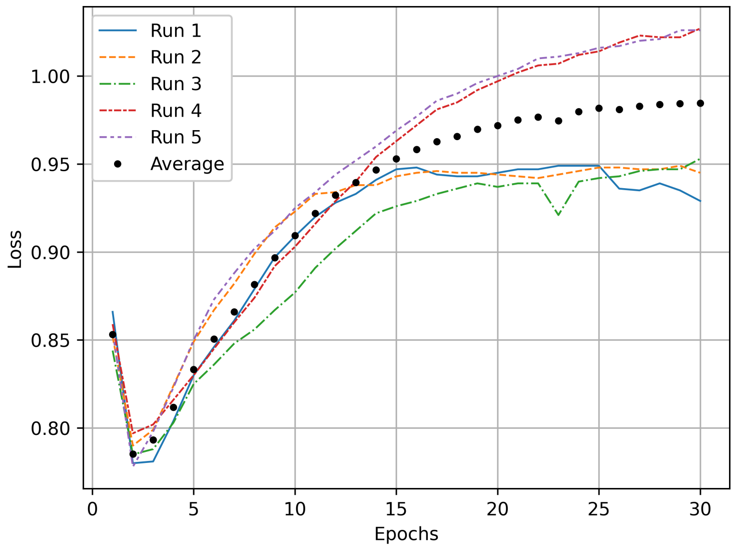


Figure 7.7: Losses of the learning model when trained on the Fashion MNIST dataset.

7.2.6. Timing results

The timings of the learning system are more complicated than those of the inference-only system. There are multiple steps that are combined in various ways to perform inference and training:

- Reading one entry from the SD card (R)
- Writing the complete input data, including the classification labels and bias, to the FPGA (D)
- Writing only the classification label inputs to the FPGA (C)
- Running inference on the FPGA (I)
- Running learning on the FPGA (L)

The last item is especially tricky, as the time required for learning depends on how well the network is trained already - after all, the algorithm forgoes the actual learning step if the loss is small enough. Thus, the average value of L varies over time.

The measured timing results can be seen in Table 7.8, and the inference and learning times are as expected. With the network calculating 13 blocks of 8 neurons for 50 blocks of 16 inputs per cycle, the system takes slightly more than 650 cycles to fully complete the inference process. At $71.4Mhz$, this takes $9.17\mu s$, which is slightly lower then the $10.05\mu s$ measured. The remaining $0.88\mu s$ is taken up by communication with the network (starting inference and checking whether it is done, and then reading the result). Similarly, the time required for learning is slightly below that of inference in the best case (as the output is not read), and approximately twice that if the weights are updated.

Although communication does take up a significant part of the time used by the network, it is still smaller than inference and learning times of the network.

At the very start the weights need to be initialised. Reading all weights from the SD card takes roughly $2ms$, and writing them was measured to take $12.8ms$, although this used a slower method than used later for writing inputs.

In practice, there are two situations, each consisting of the aforementioned steps: full inference and full learning.

In full inference, the inference is performed on all 10 labels. The data is read from the SD card and written to the FPGA. Then, inference is performed. Nine times, a different label is written to the FPGA,

Table 7.8: Time required for different steps

Step	**Time (μs)**
Read SD (R)	0.68
Write inputs (D)	63.45
Write known labels (C)	0.86
Inference (I)	10.05
Learning (L)	9.79 - 18.84

.

and for each label inference is performed and the result is read. The result is found by taking the label with the highest output. The time required for this process is $R+D+9C+10I = 172.37\mu s$, which allows inference of the training part of the dataset in $10.3s$ and of the testing part in $1.7s$.

The data transfer takes up a signification portion of this time, but as stated before, this could be greatly reduces. At a 32-bit per cycle transfer rate, the data transfer would only take an almost negligible $2.8\mu s$.

In full learning, learning is performed on both a correct and incorrect label. The data is read from the SD card and written to the FPGA. Then learning is performed, the label is changed, and learning is performed again. This results in $R + D + C + 2L = 84.58 - 102.68\mu s$. This means training an entire epoch takes only $5.07 - 6.16s$.

In this case, the data transfer time can be up to 3 times as long as the learning time, making it not just a significant portion of, but the largest part of the delay.

7.2.7. Comparison

Although the conditions are by no means equal, a comparison can be made with other implementations of the Forward-Forward algorithm. Both the software and microcontroller subteams have implemented their own versions, among which are single layer topologies with 32 neurons. A comparison between different metrics can be seen in Table 7.9.

Table 7.9: Comparison between different implementations of Forward-Forward on different devices.

Topology	32 Neurons					104 Neurons
Implementation	TF	TFLite[5]	TFLite[5]	C++[6]	Python [7]	VHDL
Platform	PC[8]	PC[8]	MCU[9]	MCU[9]	PC	FPGA
Data format	fp32	int8	int8	fp32	int8/int16	int8/int16
Achieved accuracy (%)	82.43	68.90	69.66	83.74	69.79	74.20
Epochs trained	20			10	10	4
Inference time/sample (μs)[10]	604.1	8771.5	1065.4	2533.27		100.46
Training time/epoch (s)[10]	29.24			7.12		1.17-2.26
Inference energy/sample (mJ)			0.707	1.676		0.19[11]

In terms of accuracy, the learning implemented on the FPGA clearly performs less than the other versions, despite using over 3 times the amount of neurons. However, it does outperform the quantized TensorFlow Lite inference-only versions of the model slightly. Unfortunately, this is hardly a fair comparison, as the quantized networks only use 8-bit weights and 32 neurons.

To make proper comparison of the accuracy, small test was performed of 10 runs of 10 epochs for only 32 neurons. The results can be seen in Table C.3. All but one run managed to reach at least 70%, while one run achieved no more than 62%. The highest average across the 10 epochs was 69.8%. This accuracy is comparable to that of the models quantized using TensorFlow Lite.

[5]Quantized from the full TensorFlow model.
[6]Custom implementation of regular Forward-Forward by the microcontroller subteam.
[7]Quantized implementation as used in the FPGA design.
[8]Intel(R) Xeon(R) CPU, $2.00 - 2.30MHz$, 1 core, 2 threads, $38.5 - 55MB$ L3 cache.
[9]Teensy 4.1 running at 600MHz.
[10]Not including reading the dataset.
[11]Value obtained by multiplying full estimated power by inference time excluding SD card reads.

The low accuracy mean that in its current form, the algorithm is not good enough to be useful. It does however demonstrate that the Forward-Forward algorithm can still learn when very crudely simplified. It is to be expected that implementations that stay more true to the original mathematics will perform far better, even under quantized conditions.

The system does outperform the both the computer and microcontroller implementations in terms of inference and learning time, even though it has more neurons. Full inference is $10 - 80$ times faster than the other implementations, and training is about $3 - 6$ times faster than the C++ implementation and $13 - 17$ times faster than the TensorFlow implementation. The design could definitely be improved upon to reach higher accuracy, without becoming much slower.

Although the data transfer time is certainly quite significant, even including this delay the system would be much faster than the other systems without their data transfers.

Because the system is much faster, the energy efficiency is also improved.

8 Conclusion

In conclusion, this project successfully achieved its objective of deploying a neural network architecture on the FPGA and enabling on-device learning capabilities.

The implementation process involved a study of the FPGA device, followed by a comprehensive exploration of various neural networking aspects and techniques, and their benefits and drawbacks in FPGA implementations.

Initially, an architecture designed for inference using 8-bit quantized weights was developed. The model proved to be much faster than similar networks deployed on both a computer and microcontroller.

Subsequently, the project progressed to implementing a more advanced architecture with learning capabilities on the FPGA. A simplified algorithm was proposed, based on Forward-Forward and Hebbian learning. Unfortunately, the learning algorithm proposed did not meet all accuracy requirements set by the team, although it does manage to learn to some degree and is promising enough to further investigate. At the very least, it demonstrates that even the most simplified version of the algorithm can learn, and to our knowledge no other FPGA implementations of Forward-Forward have been published yet.

Apart from one accuracy requirement, all requirements were met. Overall, the project has demonstrated the feasibility of using Forward-Forward on FPGAs for on-device learning, and highlighted the potential speed advantages of FPGA implementations.

8.1. Summary of findings

Various aspects of neural networks were discussed with regard to FPGA implementation. Design size and memory usage are both extremely important limitations of the FPGA, that shape the network design. Quantization can greatly reduce size, memory, power and time costs, but require careful scaling, can negatively influence accuracy, and makes training much harder. Parallelism is key in making effective use of FPGAs, and allow for very fast network implementations.

An 8-bit, inference only architecture was developed, and two networks for Fashion MNIST were compared to other quantized version of the same network on other machines. Because of differences in the quantization method, the accuracy achieved was slightly worse, but the inference time on the FPGA was much smaller, despite being unoptimized. This highlights the advantage of having full control over parallelisation in FPGA implementations. The neural network design can run in $14.38\mu s$ for a 784-32-10 topology excluding data transfer times, and in $15.24\mu s$ for a 784-32-32-10 topology, and speeds can be six times as high by using better pipelining and more parallel synapses, which would easily fit on the FPGA.

The suitability for FPGA implementation of the Forward-Forward and Hebbian learning algorithms were evaluated. It was found that the Forward-Forward algorithm can be seen as a method for applying Hebbian and anti-Hebbian learning rules with varying learning rates.

A simplified version of the algorithm was proposed, that uses only fixed-size weight changes, 16-bit weights and 8-bit inputs. This algorithm was implemented for a single layer on an FPGA and tested. The algorithm managed to reach 90.5% accuracy on the MNIST training set in 30 epochs, but performed considerably less well on the Fashion MNIST dataset, reaching no more than 74.2% accuracy, and declining after the first few epochs. The chosen simplification seems to be too course to be useful, but it demonstrates the robustness of the Forward-Forward learning algorithm. It is therefore thought that any more sophisticated approach should be able to reach much better results.

The FPGA implementation did again demonstrate being much faster, and thereby more energy efficient, than implementations of the Forward-Forward algorithm on digital devices, as it only used $10.05\mu s$ for inference on a single label, excluding data transfer times, and only $9.79 - 18.84\mu s$ for a single training run for one label. It is expected that even a much more sophisticated implementation on the FPGA would be considerably faster than implementations on digital devices, if the design is properly pipelined.

8.2. Limitations of the study

The study was severely limited by the time constraints of the project. Implementating systems on FPGAs takes up vastly more time than doing so in software. Because of this, only one learning algorithm could be implemented, and even then there was not enough time to fully optimize the implementation and its parameters.

8.3. Recommendations for future work

The current learning implementation is promising, but far from actually usable. Further research is needed to see if it can be improved upon. Only changing the weights by one at a time seems to be too optimistic, so perhaps slightly more accurate, but still low-precision, limited weight updates could prove to have a large effect on the accuracy achieved.

Another important change would be to include multiple layers. This requires finding a way to efficiently normalize the outputs, but has a lot of potential for improving results, as multilayered networks support higher complexity.

One problem with the design seems to be that the weights are left unbounded, and bounding them reduces the accuracy. A possible change would be to switch out the Hebbian part of the Forward-Forward equation with Grossberg's instar rule or Oja's rule. These changes should be far less drastic than using the sign function as done in the proposed learning method.

Similarly, different concepts like taking the total activation of a layer might provide suitable directives for applying Hebbian-like learning rules to a network.

Bibliography

[1] R. Lippmann, “An introduction to computing with neural nets,” *IEEE ASSP Magazine*, vol. 4, no. 2, pp. 4–22, Apr. 1987.

[2] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323. [Online]. Available: `https://proceedings.mlr.press/v15/glorot11a.html`.

[3] S. Linnainmaa, “The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors,” M.S. thesis, University of Helsinki, 1970.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[5] L. Deng, “The MNIST database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[6] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms*, 2017. arXiv: `1708.07747`.

[7] Y. Taright and M. Hubin, “FPGA implementation of a multilayer perceptron neural network using VHDL,” in *ICSP ’98. 1998 Fourth International Conference on Signal Processing (Cat. No.98TH8344)*, vol. 2, 1998, 1311–1314 vol.2. DOI: `10.1109/ICOSP.1998.770860`.

[8] I. Westby, X. Yang, T. Liu, and H. Xu, “Fpga acceleration on a multi-layer perceptron neural network for digit recognition,” *The Journal of Supercomputing*, vol. 77, no. 12, pp. 14 356–14 373, 2021.

[9] R. Gadea, J. Cerdá, F. Ballester, and A. Mocholí, “Artificial neural network implementation on a single fpga of a pipelined on-line backpropagation,” in *Proceedings of the 13th international symposium on System synthesis*, 2000, pp. 225–230.

[10] F. Ortega-Zamorano, J. M. Jerez, D. U. Munoz, R. M. Luque-Baena, and L. Franco, “Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers,” *IEEE transactions on neural networks and learning systems*, vol. 27, no. 9, pp. 1840–1850, 2015.

[11] J. Li, Y. Katori, and T. Kohno, “Hebbian learning in FPGA silicon neuronal network,” in *Proceedings of The 1st IEEE/IIAE International Conference on Intelligent Systems and Image Processing 2013*, 2013, pp. 83–90.

[12] M. Rossmann, T. Jost, K. Goser, A. Bühlmeier, and G. Manteuffel, “Exponential hebbian online learning implemented in FPGAs,” in *Artificial Neural Networks — ICANN 96*, C. von der Malsburg, W. von Seelen, J. C. Vorbrüggen, and B. Sendhoff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 767–772, ISBN: 978-3-540-68684-2.

[13] M. Rossmann, B. Hesse, K. Goser, A. Buhlmeier, and G. Manteuffel, “Implementation of a biologically inspired neuron-model in fpga,” in *Proceedings of fifth international conference on microelectronics for neural networks*, IEEE, 1996, pp. 322–329.

[14] D. Pani, P. Meloni, G. Tuveri, F. Palumbo, P. Massobrio, and L. Raffo, “An FPGA platform for real-time simulation of spiking neuronal networks,” *Frontiers in neuroscience*, vol. 11, p. 90, 2017.

[15] “Zedboard - digilent reference.” (2014), [Online]. Available: `https://digilent.com/reference/programmable-logic/zedboard/start`.

[16] D. Li, Y. Yang, Y.-Z. Song, and T. M. Hospedales, *Learning to generalize: Meta-learning for domain generalization*, 2017. arXiv: `1710.03463`.

[17] M. Moussa, S. Areibi, and K. Nichols, “On the arithmetic precision for implementing back-propagation networks on fpga: A case study,” in *FPGA Implementations of Neural Networks*, A. R. Omondi and J. C. Rajapakse, Eds. Boston, MA: Springer US, 2006, pp. 37–61, ISBN: 978-0-387-28487-3. DOI: 10.1007/0-387-28487-7_2. [Online]. Available: https://doi.org/10.1007/0-387-28487-7_2.

[18] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” *CoRR*, vol. abs/1603.05279, 2016. arXiv: 1603.05279. [Online]. Available: http://arxiv.org/abs/1603.05279.

[19] P. Yin, J. Lyu, S. Zhang, S. Osher, Y. Qi, and J. Xin, *Understanding straight-through estimator in training activation quantized neural nets*, 2019. arXiv: 1903.05662.

[20] K. Helwegen, J. Widdicombe, L. Geiger, Z. Liu, K.-T. Cheng, and R. Nusselder, *Latent weights do not exist: Rethinking binarized neural network optimization*, 2019. arXiv: 1906.02107.

[21] A. R. Omondi, J. C. Rajapakse, and M. Bajger, “FPGA neurocomputers,” in *FPGA Implementations of Neural Networks*, A. R. Omondi and J. C. Rajapakse, Eds. Boston, MA: Springer US, 2006, pp. 1–36, ISBN: 978-0-387-28487-3. DOI: 10.1007/0-387-28487-7_1.

[22] H. Noh, T. You, J. Mun, and B. Han, *Regularizing deep neural networks by noise: Its interpretation and optimization*, 2017. arXiv: 1710.05179.

[23] M. Höhfeld and S. E. Fahlman, “Probabilistic rounding in neural network learning with limited precision,” *Neurocomputing*, vol. 4, no. 6, pp. 291–299, 1992, ISSN: 0925-2312. DOI: 10.1016/0925-2312(92)90014-G. [Online]. Available: https://www.sciencedirect.com/science/article/pii/092523129290014G.

[24] M. Kang, “Fpga implementation of gaussian-distributed pseudo-random number generator,” in *6th International Conference on Digital Content, Multimedia Technology and its Applications*, 2010, pp. 11–13.

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

[26] D. O. Hebb, *The Organization of Behavior*. New York: Wiley, 1949.

[27] M. Gupta, S. K. Modi, H. Zhang, J. H. Lee, and J. H. Lim, *Is bio-inspired learning better than backprop? Benchmarking bio learning vs. backprop*, 2023. arXiv: 2212.04614.

[28] S. Grossberg, “Adaptive pattern classification and universal recoding: I. parallel development and coding of neural feature detectors,” *Biological Cybernetics*, vol. 23, no. 3, pp. 121–134, 1976. DOI: 10.1007/bf00344744.

[29] E. Oja, “Simplified neuron model as a principal component analyzer,” *Journal of Mathematical Biology*, vol. 15, no. 3, pp. 267–273, Nov. 1982. DOI: 10.1007/bf00275687.

[30] R. W. Strom, “Hebbian Learning in Multilayer Neural Networks,” M.S. thesis, California State University, Los Angeles, 2007.

[31] P. Dayan, M. Sahani, and G. Deback, “Unsupervised learning,” *The MIT encyclopedia of the cognitive sciences*, pp. 857–859, 1999.

[32] K. You, M. Long, J. Wang, and M. I. Jordan, *How does learning rate decay help modern neural networks?* 2019. arXiv: 1908.01878.

[33] J. Heaton, “Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning,” *Genetic Programming and Evolvable Machines*, vol. 19, no. 1, pp. 305–307, Jun. 2018, ISSN: 1573-7632. DOI: 10.1007/s10710-017-9314-z. [Online]. Available: https://doi.org/10.1007/s10710-017-9314-z.

[34] R. Rojas, “The backpropagation algorithm,” in *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 149–182, ISBN: 978-3-642-61068-4. DOI: 10.1007/978-3-642-61068-4_7. [Online]. Available: https://doi.org/10.1007/978-3-642-61068-4_7.

[35] Google Brain Team, *Tensorflow*, 2023. [Online]. Available: https://www.tensorflow.org/.

[36] S. A. AmirReza Rajabi, *Tensorflow-Forward-Forward*, 2023. [Online]. Available: `https://github.com/amirrezarajabi/Tensorflow-Forward-Forward`.

[37] Y. Mohammad Pezeshki Haaris Rahman, *pytorch_forward_forward*, 2020. [Online]. Available: `https://github.com/mohammadpz/pytorch_forward_forward`.

[38] S. Mukherjee, *Keras documentation: Using the forward-forward algorithm for image classification*, Aug. 2023. [Online]. Available: `https://keras.io/examples/vision/forwardforward/`.

A

Mathematics of learning algorithms

A.1. Backpropagation

This section covers the mathematics behind the backpropagation algorithm.

For outputs x_i of one layer, outputs y_j of the next layer, and respective pre-activation function sums ψ_i and ψ_j, connecting weights w_{ij}, activation function $f(\psi)$ and loss $\mathcal{L}$ that depends on the output of the network, we see that:

$$\frac{\partial \mathcal{L}}{\partial \psi_j} = f'(\psi_j) \cdot \frac{\partial \mathcal{L}}{\partial y_j}, \ \frac{\partial \mathcal{L}}{\partial \psi_i} = f'(\psi_i) \cdot \frac{\partial \mathcal{L}}{\partial x_i} \tag{A.1}$$

$$\frac{\partial \mathcal{L}}{\partial x_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \psi_j} \cdot w_{ij} \tag{A.2}$$

Combing these, we find that the gradients of the loss are propagated through the network according to:

$$\frac{\partial \mathcal{L}}{\partial x_i} = \sum_j \frac{\partial \mathcal{L}}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial \mathcal{L}}{\partial y_j} \cdot \frac{\partial y_j}{\partial \psi_j} \cdot \frac{\partial \psi_j}{\partial x_i} = \sum_j \frac{\partial \mathcal{L}}{\partial y_j} \cdot f'(\psi_j) \cdot w_{ij} = \sum_j \frac{\partial \mathcal{L}}{\partial \psi_j} \cdot w_{ij} \tag{A.3}$$

We can also see that the gradients of the loss with respect to the weights are:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial \psi_j} \cdot x_i = \frac{\partial \mathcal{L}}{\partial y_j} \cdot f'(\psi_j) \cdot x_i \tag{A.4}$$

Here $\frac{\partial \mathcal{L}}{\partial \psi_j}$ is called the *local loss*. This term can be found in both equations, and only has to be calculated once per neuron j, as it is irrespective of i.

The loss gradient in the final layer is determined by the loss function. For the mean squared error of an output layer of N neurons with actual outputs z_k and desired outputs z'_k, we find:

$$\mathcal{L} = \frac{1}{N} \sum_k (z_k - z'_k)^2 \tag{A.5}$$

and

$$\frac{\partial \mathcal{L}}{\partial z_k} = \frac{2}{N} (z_k - z'_k) \tag{A.6}$$

A.2. Forward-Forward

This section covers the mathematics behind the Forward-Forward algorithm, based on [25] and several existing implementations.

Let the output y_j of neuron j be the result of activation function $f(\psi)$:

$$y_j = f(\psi_j) \tag{A.7}$$

where ψ is the sum of the products of the weights w_{ij} and inputs x_i:

$$\psi_j = \sum_i w_{ij} x_i \tag{A.8}$$

as described in subsection 2.1.1. Various implementations [36]–[38] then suggests the total and average activity of a layer of N neurons is calculated as:

$$\phi = \sum_{j=1}^{N} \left(y_j^2 - \bar{\theta}\right),\ \bar{\phi} = \phi / N \tag{A.9}$$

$\bar{\theta}$ is an offset that [25] suggests takes on a value of 2. The weights are updated based on a weight change Δw:

$$w_{ij} \coloneqq w_{ij} + \Delta w_{ij} \tag{A.10}$$

The losses for correctly labeled inputs $\mathcal{L}_{pos}$ and incorrectly labeled inputs $\mathcal{L}_{neg}$ are then defined as[1]:

$$\mathcal{L}_{pos} = N \cdot \ln\left(1 + e^{-\bar{\phi}}\right),\ \mathcal{L}_{neg} = N \cdot \ln\left(1 + e^{\bar{\phi}}\right) \tag{A.11}$$

and the weight change is found using the gradient of the loss and learning rate η:

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{L}}{\partial w_{ij}} = -\eta \frac{\partial \mathcal{L}}{\partial \bar{\phi}} \cdot \frac{\partial \bar{\phi}}{\partial y_j} \cdot \frac{\partial y_j}{\partial \psi_j} \cdot \frac{\partial \psi_j}{\partial w_{ij}} \tag{A.12}$$

We can see that:

$$\frac{\partial \mathcal{L}_{pos}}{\partial \bar{\phi}} = -N \cdot \frac{1}{1 + e^{\bar{\phi}}} = -N \cdot \mathrm{sig}(-\bar{\phi}),\ \frac{\partial \mathcal{L}_{neg}}{\partial \bar{\phi}} = N \cdot \frac{1}{1 + e^{-\bar{\phi}}} = N \cdot \mathrm{sig}(\bar{\phi}) \tag{A.13}$$

$$\frac{\partial \bar{\phi}}{\partial y_j} = \frac{2 y_j}{N} \tag{A.14}$$

$$\frac{\partial y_j}{\partial \psi_j} = \frac{\partial f(\psi)}{\partial \psi} \tag{A.15}$$

$$\frac{\partial \psi_j}{\partial w_{ij}} = x_i \tag{A.16}$$

If we use the ReLU activation function, we get:

$$f(\psi) = \mathrm{ReLU}(\psi) = \begin{cases} \psi, & \text{if } \psi > 0 \\ 0, & \text{otherwise} \end{cases} \tag{A.17}$$

$$\frac{\partial f(\psi)}{\partial \psi} = \begin{cases} 1, & \text{if } \psi > 0 \\ 0, & \text{if } \psi < 0 \end{cases} \tag{A.18}$$

and since:

$$y_j \cdot \frac{\partial y_j}{\partial \psi_j} = \begin{cases} \psi_j, & \text{if } \psi_j > 0 \\ 0, & \text{otherwise} \end{cases} \cdot \begin{cases} 1, & \text{if } \psi_j > 0 \\ 0, & \text{if } \psi_j < 0 \end{cases} = \begin{cases} \psi_j, & \text{if } \psi_j > 0 \\ 0, & \text{otherwise} \end{cases} = y_j \tag{A.19}$$

we finally find:

$$\begin{aligned} \Delta w_{ij,\ pos} &= \eta \mathrm{sig}(-\bar{\phi}) \cdot 2 x_i y_j \\ \Delta w_{ij,\ neg} &= -\eta \mathrm{sig}(\bar{\phi}) \cdot 2 x_i y_j \end{aligned} \tag{A.20}$$

Since both x and y are non-negative, $\Delta w_{ij,\ pos} \geq 0$ and $\Delta w_{ij,\ neg} \leq 0$.

[1] Although the original paper uses the absolute sum of the squared outputs ϕ, many online implementations [36]–[38] use the mean value $\bar{\phi}$ instead. However, the scaling factor $1/N$ is not used in the derivative of $\mathcal{L}$. This prompts us to include an extra scaling factor N in the definition of $\mathcal{L}$ - the version without this factor, which is commonly used as a measure of training progress, can then be considered the "average loss" ($\bar{\mathcal{L}}$) instead. Ultimately, the choice between total and average values is one of scaling, and only affects the sizes of values and which values remain the same when the number of neurons is changed.

B Scaling

Table B.1: Distributions of numbers requiring a certain number of bits at the unscaled outputs of the different layers in both inference topologies. The 8 most significant bits are marked.

	784-32-10		784-32-32-10		
Bits	**L2**	**L3**	**L2**	**L3**	**L4**
1	0	9	3	20	3
2	2	15	2	36	11
3	7	27	3	71	16
4	8	55	17	156	36
5	18	131	26	259	69
6	32	234	53	511	146
7	65	472	109	1032	271
8	169	1022	222	1987	535
9	292	1978	446	4174	1107
10	581	3845	875	8229	2351
11	1112	7615	1683	16067	4636
12	2130	13385	3529	30981	8580
13	4324	13569	6946	52451	12410
14	8305	8109	13366	57720	13240
15	16194	534	23939	20446	7134
16	28857		38402	1704	274
17	43353		42807		
18	39188		22157		
19	14337		1564		
20	699				

C Results

C.1. Learning model accuracies

Table C.1: Accuracies of the test data for different runs of the learning algorithm on the MNIST dataset. Maximum values are marked in bold.

Epoch	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average
1	0.8402	0.8473	0.8505	0.8394	0.8372	0.8194	0.8524	0.8519	0.8298	0.8594	0.8428
2	0.8609	0.8690	0.8708	0.8644	0.8666	0.8532	0.8689	0.8721	0.8580	0.8820	0.8666
3	0.8790	0.8809	0.8792	0.8796	0.8830	0.8703	0.8804	0.8842	0.8771	0.8899	0.8804
4	0.8877	0.8898	0.8833	0.8916	0.8908	0.8811	0.8894	0.8889	0.8860	0.8930	0.8882
5	0.8960	0.8939	0.8902	0.8904	0.8963	0.8806	0.8945	0.8919	0.8938	0.8945	0.8922
6	0.8974	0.8938	0.8879	0.8900	0.8965	0.8924	0.8957	0.8958	0.8978	0.8956	0.8943
7	0.8974	0.8967	0.8908	0.8980	0.9017	0.8919	0.8986	0.8973	0.9020	0.8982	0.8973
8	0.9014	0.8976	0.8957	0.9019	0.9016	0.8956	0.8993	0.8976	0.9037	0.9000	0.8994
9	0.9011	0.8998	0.9015	0.8999	0.9042	0.8970	0.9039	0.8976	0.9057	0.9002	0.9011
10	0.9008	0.903	0.9001	0.9001	0.9036	0.8957	0.9024	0.8985	**0.9058**	0.9000	0.9010
11	**0.9014**	0.8994	0.9054	0.8997	0.9038	0.8972	0.9030	0.8991	0.9047	0.9037	0.9017
12	0.8983	0.9026	0.9057	0.8996	0.9038	0.8963	0.9023	0.8985	0.9053	0.9048	0.9017
13	0.8965	0.9005	0.9090	0.9026	0.9038	0.8975	0.9030	0.9001	0.9010	0.9030	0.9017
14	0.8961	0.9001	0.9096	0.9020	0.9052	0.8986	0.9032	0.8996	0.9034	0.9049	0.9023
15	0.8967	0.9005	0.9111	0.9043	0.9066	0.8979	0.9029	0.9008	0.9034	0.9059	0.9030
16	0.8977	0.9037	0.9120	0.9031	0.9075	0.8997	0.9035	0.9017	0.9034	0.9047	0.9037
17	0.8980	0.9008	**0.9141**	0.9011	0.9062	0.8963	0.9053	0.8991	0.9036	0.9051	0.9030
18	0.8984	0.9025	0.9131	0.9036	0.9081	0.9001	0.9069	0.9018	0.9029	0.9055	0.9043
19	0.8981	0.9045	0.9118	0.9052	0.9079	0.8980	0.9076	0.8987	0.9008	0.9061	0.9039
20	0.8981	0.9038	0.9121	0.9052	0.9062	0.8962	0.9056	0.9004	0.9015	0.9061	0.9035
21	0.9012	0.9044	0.9101	**0.9059**	0.9099	0.8963	**0.9078**	0.9001	0.8989	**0.9067**	0.9041
22	0.8994	0.9041	0.9112	0.9055	0.9112	0.8985	0.9057	0.9020	0.9005	0.9062	0.9044
23	0.8992	0.9038	0.9124	0.9036	0.9099	0.8984	0.9042	0.9020	0.9002	0.9045	0.9038
24	0.8983	0.9039	0.9111	0.9047	0.9127	0.9006	0.9054	0.9014	0.9003	0.9065	0.9045
25	0.8981	0.9038	0.9122	0.9038	0.9132	0.9013	0.9030	0.9016	0.9008	0.9052	0.9043
26	0.8996	0.9054	0.9124	0.9054	0.9138	0.8999	0.9057	0.9019	0.9014	0.9052	0.9051
27	0.8995	0.9060	0.9113	0.9038	0.9140	0.9008	0.9026	0.9013	0.9019	0.9060	0.9047
28	0.8987	0.9044	0.9096	0.9055	0.9139	0.9019	0.9030	0.9021	0.9035	0.9054	0.9048
29	0.8995	**0.9059**	0.9098	0.9043	**0.9150**	**0.9042**	0.9031	0.9024	0.9038	0.9060	**0.9054**
30	0.8966	0.9058	0.9092	0.9038	0.9142	0.9027	0.9033	**0.9030**	0.9045	0.9056	0.9049
Max	0.9014	0.9060	0.9141	0.9059	0.9150	0.9042	0.9078	0.9030	0.9058	0.9067	0.9054

C.2. Learning model losses

Table C.2: Accuracies of the test data for different runs of the learning algorithm on the Fashion MNIST dataset. Maximum values are marked in bold.

Epoch	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Average**
1	0.7153	0.6992	0.6860	0.7100	0.7070	0.7001	0.7029
2	0.7220	0.7349	0.6925	0.7318	0.7188	0.7138	0.7190
3	0.7359	0.7557	0.7036	0.7484	0.7215	0.7261	0.7319
4	0.7332	**0.7558**	0.7425	**0.7485**	0.7226	0.7496	**0.7420**
5	0.7360	0.7324	0.7418	0.7192	0.7206	0.7350	0.7308
6	0.7403	0.7351	**0.7548**	0.7195	**0.7298**	**0.7573**	0.7395
7	0.7396	0.7375	0.7517	0.7209	0.7287	0.7480	0.7377
8	0.7416	0.7437	0.7496	0.7199	0.7298	0.7556	0.7400
9	0.7423	0.7427	0.7485	0.7191	0.7250	0.7573	0.7392
10	0.7466	0.7442	0.7482	0.7243	0.7222	0.7350	0.7368
11	0.7450	0.7423	0.7452	0.7216	0.7249	0.7313	0.7351
12	0.7497	0.7432	0.7470	0.723	0.7261	0.7250	0.7357
13	0.7507	0.7442	0.7449	0.7254	0.7246	0.7231	0.7355
14	**0.7523**	0.7445	0.7456	0.7251	0.7237	0.7203	0.7353
15	0.7496	0.7433	0.7444	0.7245	0.7238	0.7197	0.7342
16	0.7476	0.7424	0.7393	0.7256	0.7209	0.7217	0.7329
17	0.7477	0.7429	0.7388	0.7263	0.7191	0.7183	0.7322
18	0.7457	0.7431	0.7361	0.7281	0.7191	0.7190	0.7319
19	0.7465	0.7406	0.7371	0.7260	0.721	0.7185	0.7316
20	0.7468	0.7379	0.7371	0.7198	0.7182	0.7156	0.7292
21	0.7461	0.7403	0.7374	0.7263	0.7155	0.7132	0.7298
22	0.7411	0.7370	0.7342	0.7184	0.7156	0.7118	0.7264
23	0.7424	0.7387	0.7195	0.7172	0.7130	0.7112	0.7237
24	0.7388	0.7380	0.7290	0.7167	0.7112	0.7118	0.7243
25	0.7383	0.7364	0.7274	0.7163	0.7115	0.7100	0.7233
26	0.7347	0.7342	0.7270	0.7157	0.7089	0.7095	0.7217
27	0.7350	0.7351	0.7251	0.7137	0.7093	0.7072	0.7209
28	0.7331	0.7366	0.7207	0.7154	0.7086	0.7069	0.7202
29	0.7200	0.7356	0.7186	0.7127	0.7064	0.7055	0.7165
30	0.7222	0.7340	0.7206	0.7131	0.7035	0.7033	0.7161
Max	0.7523	0.7558	0.7548	0.7485	0.7298	0.7573	0.7420

Table C.3: Accuracies of the test data for different runs of the learning algorithm on the Fashion MNIST dataset when using only 32 neurons.

Epoch	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Max**
Run 1	0.6644	0.6867	0.6929	0.6995	0.7046	0.7049	0.7032	0.7051	0.702	0.7022	0.7051
Run 2	0.6152	0.5993	0.6093	0.6156	0.6162	0.6153	0.6138	0.6127	0.6090	0.6105	0.6162
Run 3	0.5562	0.6795	0.6954	0.7050	0.7039	0.7059	0.7077	0.7148	0.7124	0.7130	0.7148
Run 4	0.5521	0.6203	0.6417	0.6909	0.6968	0.7015	0.7018	0.6995	0.7000	0.6973	0.7018
Run 5	0.6716	0.6942	0.7045	0.7103	0.7084	0.7049	0.7099	0.7038	0.7078	0.7058	0.7103
Run 6	0.6559	0.6944	0.7032	0.7130	0.7177	0.7159	0.7176	0.7105	0.7064	0.7148	0.7177
Run 7	0.5744	0.6103	0.6385	0.6943	0.7063	0.7092	0.7096	0.7144	0.7195	0.7216	0.7216
Run 8	0.6410	0.6804	0.6842	0.6977	0.7017	0.7075	0.7032	0.7005	0.7035	0.7041	0.7075
Run 9	0.6450	0.6625	0.7003	0.6970	0.7004	0.704	0.7090	0.7077	0.7063	0.7049	0.7090
Run 10	0.6110	0.6787	0.6817	0.7060	0.7073	0.7059	0.7021	0.7085	0.7037	0.7045	0.7085
Average	0.6187	0.6606	0.6752	0.6929	0.6963	0.6975	0.6978	0.6978	0.6971	0.6979	0.6979

Table C.4: Losses of the test data for different runs of the learning algorithm on the MNIST dataset. Minimum values are marked in bold.

Epoch	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**	**Run 9**	**Run 10**	**Average**
1	0.720	0.697	0.708	0.703	0.705	0.716	0.713	0.692	0.717	0.693	0.7064
2	0.644	0.600	0.617	0.616	0.613	0.617	0.624	0.616	0.641	0.604	0.6192
3	0.629	0.585	0.603	0.601	0.593	0.600	0.603	0.608	0.619	0.596	0.6037
4	0.621	0.578	0.598	0.596	0.585	0.595	0.595	0.603	0.612	0.596	0.5979
5	0.617	0.575	0.595	0.594	0.580	0.591	0.593	0.602	0.603	0.595	0.5945
6	0.615	0.575	0.592	0.593	0.576	0.589	**0.590**	0.600	0.592	0.595	0.5917
7	0.611	0.574	0.591	0.591	0.574	0.588	0.591	0.598	0.590	0.594	0.5902
8	0.610	0.574	0.590	**0.590**	**0.574**	0.588	0.591	0.596	**0.589**	0.593	0.5895
9	0.608	**0.573**	**0.589**	0.591	0.574	**0.588**	0.594	0.595	0.590	0.590	0.5892
10	0.601	0.576	0.591	0.593	0.575	0.588	0.594	0.593	0.592	0.590	0.5893
11	**0.599**	0.576	0.590	0.595	0.575	0.589	0.594	0.591	0.591	0.589	**0.5889**
12	0.600	0.578	0.590	0.596	0.576	0.589	0.595	**0.591**	0.593	**0.589**	0.5897
13	0.603	0.579	0.590	0.599	0.577	0.590	0.596	0.591	0.594	0.591	0.591
14	0.606	0.578	0.591	0.600	0.579	0.591	0.597	0.592	0.595	0.590	0.5919
15	0.608	0.579	0.591	0.601	0.581	0.593	0.598	0.593	0.596	0.591	0.5931
16	0.611	0.580	0.590	0.603	0.583	0.593	0.599	0.595	0.598	0.593	0.5945
17	0.612	0.580	0.591	0.605	0.584	0.595	0.598	0.596	0.599	0.595	0.5955
18	0.616	0.584	0.590	0.606	0.585	0.597	0.599	0.596	0.600	0.596	0.5969
19	0.618	0.585	0.592	0.608	0.586	0.598	0.601	0.597	0.602	0.597	0.5984
20	0.620	0.587	0.591	0.606	0.587	0.601	0.602	0.599	0.603	0.599	0.5995
21	0.623	0.587	0.592	0.606	0.587	0.601	0.603	0.600	0.604	0.601	0.6004
22	0.626	0.589	0.593	0.608	0.589	0.603	0.605	0.601	0.605	0.604	0.6023
23	0.628	0.592	0.595	0.610	0.590	0.605	0.606	0.603	0.606	0.606	0.6041
24	0.631	0.594	0.595	0.610	0.591	0.607	0.608	0.605	0.608	0.607	0.6056
25	0.634	0.595	0.598	0.613	0.593	0.609	0.610	0.607	0.610	0.609	0.6078
26	0.636	0.597	0.599	0.614	0.595	0.611	0.612	0.609	0.610	0.610	0.6093
27	0.639	0.600	0.600	0.616	0.596	0.612	0.614	0.613	0.612	0.611	0.6113
28	0.642	0.602	0.602	0.619	0.598	0.615	0.616	0.614	0.613	0.611	0.6132
29	0.644	0.605	0.604	0.621	0.600	0.616	0.618	0.616	0.614	0.612	0.6150
30	0.646	0.608	0.605	0.622	0.600	0.618	0.620	0.618	0.616	0.614	0.6167
Min	0.599	0.573	0.589	0.590	0.574	0.588	0.590	0.591	0.589	0.589	0.5889

Table C.5: Losses of the test data for different runs of the learning algorithm on the Fashion MNIST dataset. Minimum values are marked in bold.

Epoch	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Average
1	0.866	0.852	0.844	0.859	0.855	0.843	0.8532
2	**0.780**	**0.79**	**0.785**	**0.797**	**0.778**	**0.781**	**0.7852**
3	0.781	0.799	0.788	0.802	0.798	0.791	0.7932
4	0.804	0.824	0.803	0.816	0.823	0.801	0.8118
5	0.830	0.849	0.825	0.830	0.850	0.816	0.8333
6	0.846	0.867	0.836	0.845	0.873	0.836	0.8505
7	0.861	0.882	0.848	0.860	0.888	0.857	0.8660
8	0.879	0.899	0.856	0.874	0.902	0.879	0.8815
9	0.897	0.914	0.867	0.892	0.912	0.899	0.8968
10	0.909	0.923	0.877	0.903	0.925	0.919	0.9093
11	0.920	0.933	0.891	0.916	0.934	0.938	0.9220
12	0.928	0.934	0.902	0.929	0.944	0.957	0.9323
13	0.933	0.938	0.912	0.940	0.952	0.962	0.9395
14	0.941	0.938	0.922	0.954	0.960	0.965	0.9467
15	0.947	0.943	0.926	0.963	0.969	0.970	0.9530
16	0.948	0.945	0.929	0.972	0.977	0.979	0.9583
17	0.944	0.946	0.933	0.981	0.986	0.986	0.9627
18	0.943	0.945	0.936	0.985	0.990	0.995	0.9657
19	0.943	0.945	0.939	0.992	0.996	1.004	0.9698
20	0.945	0.944	0.937	0.997	1.000	1.008	0.9718
21	0.947	0.943	0.939	1.002	1.004	1.015	0.9750
22	0.947	0.942	0.939	1.006	1.010	1.016	0.9767
23	0.949	0.944	0.921	1.007	1.011	1.015	0.9745
24	0.949	0.946	0.940	1.012	1.013	1.019	0.9798
25	0.949	0.948	0.942	1.014	1.016	1.021	0.9817
26	0.936	0.948	0.943	1.019	1.017	1.023	0.9810
27	0.935	0.947	0.946	1.023	1.020	1.026	0.9828
28	0.939	0.947	0.947	1.022	1.021	1.027	0.9838
29	0.935	0.949	0.947	1.022	1.026	1.027	0.9843
30	0.929	0.945	0.953	1.027	1.026	1.027	0.9845
Min	0.780	0.790	0.785	0.797	0.778	0.781	0.7852