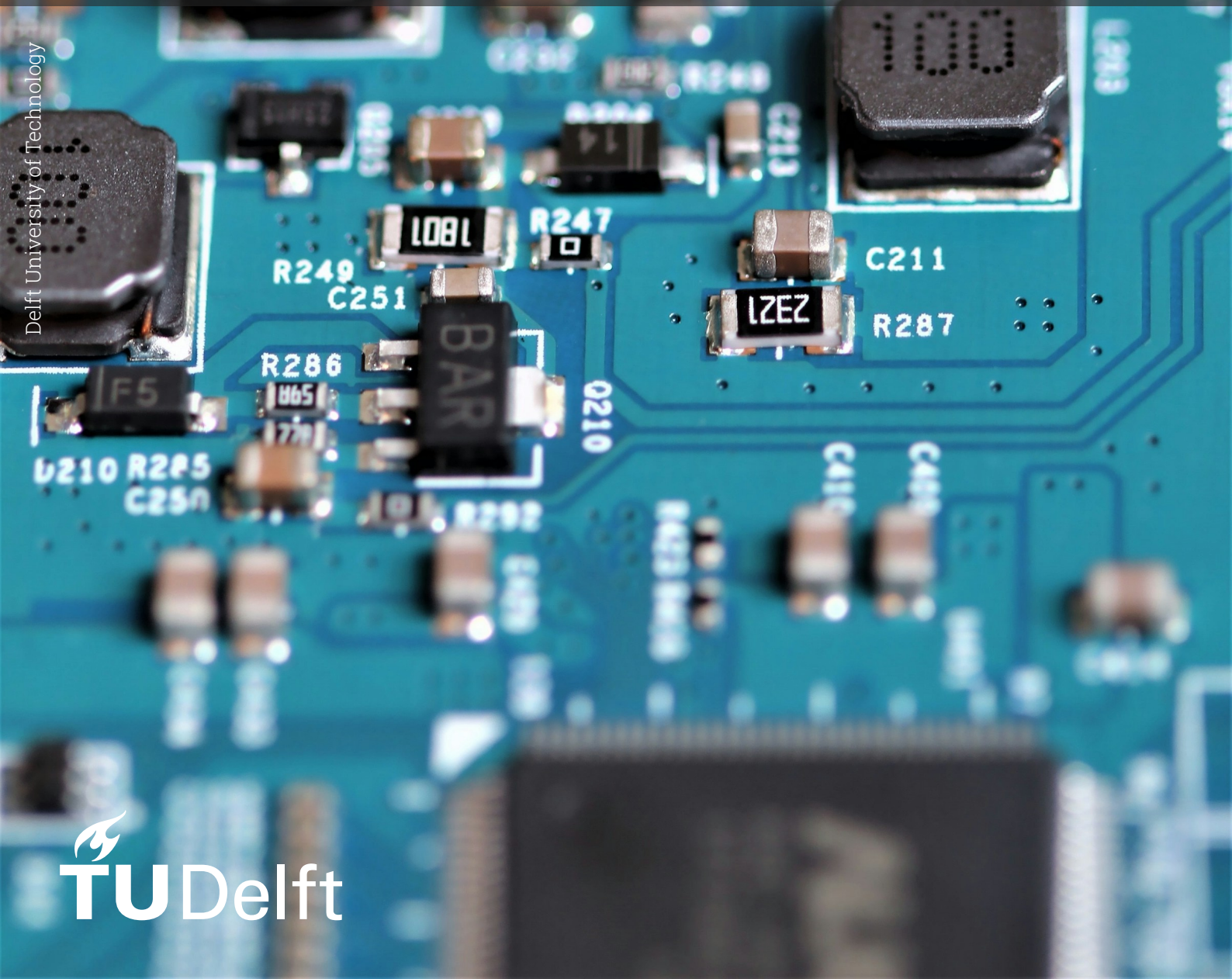
Self-Supervised Federated Learning at the Edge

Hardware and System Development

Bachelor Thesis

N. Beladel & I.R.P.T.J van Ingen



Delft University of Technology

TU Delft

Self-Supervised Federated Learning at the Edge

Hardware and System Development

by

N. Beladel & I.R.P.T.J van Ingen

Student Name	Student Number
Nassim Beladel	5644216
Ian van Ingen	5638704

Supervisors:	Dr. C. Frenkel	TU Delft	EI
	Dr. J.H.G. Dauwels	TU Delft	SPS
Project Assistant:	D. Casnici	TU Delft	EI
Thesis committee:	Dr. C. Frenkel	TU Delft	EI
	Dr. J.H.G. Dauwels	TU Delft	SPS
	Prof. Dr. N. Llombart	TU Delft	THZ
	Dr. P Manganiello	TU Delft	PMD
Thesis defence date:	27th of June, 2024		
Faculty:	EEMCS		
Degree:	BSc Electrical Engineering		

TU Delft

Preface

This thesis serves to finalise the bachelor graduation project on the topic of self-supervised federated learning, specifically the on-chip implementation of the algorithms. The goal of the project is to implement a self-supervised learning setup in a decentralised approach using Field-Programmable Gate Arrays (FPGAs) for the processing of data. In this thesis, we endeavour to illustrate the possibility of employing FPGAs to move the fairly compute-intensive self-supervised learning algorithms to the edge. We have developed a number of modules that can accelerate key algorithmic blocks that underlie the major bottlenecks of the classical application of the algorithms and showcase prospective results, which are extensively discussed afterwards to pave a clear path towards truly autonomous and efficient edge-intelligence.

N. Beladel & I.R.P.T.J van Ingen
Delft, July 2024

Contents

1 Introduction

With the increasing presence of unharnessed compute at the edge and the exponential growth of (unlabeled) datastreams, the conventional centralised approach to machine learning is starting to face its limitations. This thesis project endeavours to explore self-supervised federated learning at the edge, an up-and-coming field that could effectively address these challenges. The convergence of edge-computing and federated learning, namely, promises to address critical concerns surrounding privacy, latency, and bandwidth by distributing machine learning tasks across decentralised nodes. Enhancing these models with self-supervised machine learning capabilities promises to address the lack of labeled data, abstracting useful information from the huge, otherwise off-limit datastreams at the edge.

However, self-supervised learning models incur insurmountable computational overhead, translating to extreme latency and power requirements. Traditional applications employing a centralised approach showcase these extreme requirements clearly: self-supervised training of a fairly simple image-classifier already requires hours of training on supercomputer clusters [1]. These limitations obstruct the move to the edge, and underline the necessity of federating training and inference and reveal the potential advantage of hardware-acceleration.

Inspired by [2], proposing the combination of a self-supervised learning algorithm with a federated learning setup to tackle precisely the problem of computational overhead, we endeavour to explore the convergence of these paradigms applied to an on-chip implementation, reflecting its practical application. Due to their easy reconfigurability and use-case specific adaptation, Field-Programmable Gate Arrays, or FPGAs, provide the perfect platform to explore these possibilities and can pave the way towards truly autonomous and efficient edge-intelligence.

1.1. Problem Definition

In lieu of the scope of this endeavour, we have split responsibilities among two subgroups, respectively addressing the software and simulation framework and the actual on-chip implementation. This thesis corresponds to the latter challenge, delving into the deployment and acceleration of the learning algorithms. More generally, the subgroup division can be described as follows:

- **Hardware and System Development subgroup**. Responsible for the hardware acceleration of the self-supervised learning setup on the FPGA.
- **Signal Processing and Algorithms subgroup**. Responsible for the implementation of the federated self-supervised learning setup in Python, simulating the constraints of an edge-device.

The remainder of this chapter will be dedicated to delineating some motivational details from which the rest of this thesis departs as well as its outline.

1.2. FPGAs at the Edge

As computational resources available at the edge continue to increase, while the volume of edge-generated datastreams similarly takes off, the prospect of moving computational processes (e.g., as those employed in machine learning tasks) to the edge becomes both more feasible and more attractive. Moreover, in today's rapidly evolving technological landscape, the prospect of real-time dataprocessing, low latency applications, yet concurrently the promise of cheap deployment with minimal footprint experience growing demand. Considering these dichotomous trends, classical general-purpose computer architectures, such as the Central Processing Unit (CPU) or the Graphics Processing Unit (GPU), can no longer completely satisfy consumer and application-driven needs. Conversely, FPGAs house easily reconfigurable hardware components that can 'emulate' application-specific chip designs and thus embed dedicated acceleration modules for any given computational task. Modern development boards usually include both a traditional FPGA and an on-board microcontroller, as well as a wide range of interfaces to facilitate design and deployment, yet remain within very affordable regimes for low volume applications. The on-board FPGA can be used to offload computationally demanding tasks from the CPU or GPU to dedicated modules, moreover decreasing footprint by maximising efficient hardware use. FPGA designs can specifically target even lower power usages, if so desired, through hardware reuse at the cost of latency, further contributing to minimising footprint – or, conversely, exploit the massive parallelism afforded by the design of dedicated hardware. These opportunities make FPGAs a worthwhile avenue to explore and exploit in the move to the edge.

1.3. Thesis Outline

This thesis will elaborate on the hardware acceleration on the client nodes that embed a down-scaled version of the algorithms implemented by the Signal Processing and Algorithms subgroup. Chapter 2 will discuss the requirements of the entire system and the requirements specific to the client nodes that we have set forth. We will provide the requisite theoretical background for the remainder of the thesis in the following chapter, Chapter 3. Chapter 4 delves deeper into the computational requirements of the algorithms after which a basic model is evaluated to identify the major bottlenecks. Chapter 5 discusses the acceleration modules that have been designed to offload computationally intensive tasks of the algorithm from the microprocessor to the programmable logic. The communication between the microprocessor and programmable logic, including the transfer of data and a microprocessor interface allowing control of the programmable logic, is described in Chapter 6 together with a design overview of the interconnections between all designed modules. Chapter 7 shows the performance of the hardware modules that have been implemented and showcases the execution times of the necessary functionalities on the FPGA and microprocessor. These results and a perspective on future developments will be discussed in Chapter 8 after which a brief conclusion in Chapter 9 will mark the end of this thesis.

2

Programme of Requirements

2.1. Requirements Proof-of-Concept

The goal of the entire project group is to design a proof-of-concept system that combines self-supervised learning and federated learning on an FPGA. This concept can then be modified in the future to be more applicable to specific cases in which such a set-up is required. The entire system must satisfy a number of main requirements:

1. The system must be able to run a self-supervised learning algorithm on a client node.
2. The system must allow for the federation of training through a federated learning set-up.
3. The system must be scalable to allow for multiple client nodes, with a minimum number of two client nodes.
4. The system's algorithms must be accelerated by using the programmable logic available on the client FPGAs.

2.2. Requirements Hardware and System Development

The Hardware and System Development subgroup's main responsibilities are to implement the algorithms made by the Signal Processing and Algorithms subgroup on the client nodes, consisting of a classical microprocessor and programmable logic, and to design and implement hardware acceleration modules to accelerate these algorithms. Based on these tasks, a number of additional requirements can be compiled.

Some of these requirements are mandatory requirements; these are requirements that should be satisfied to prove the main positions set forth by this thesis. They mainly refer to what must be achieved to satisfy the main requirements described above (1-6) and to the available resources on the development board (7-9, see Section 6.1):

1. The system must be able to run a self-supervised learning algorithm.
2. The system must be able to share data to an external server.
3. The system must only send model parameters to the server and no other data.
4. The system must have hardware modules implemented in the programmable logic to accelerate key algorithmic blocks.
5. The system must be able to transfer data between the microprocessor and programmable logic.
6. The system must be able to control the modules in the programmable logic via the microprocessor.
7. The system must utilize less than 144 Block RAM blocks in the programmable logic.
8. The system must utilize less than 234k flipflops in the programmable logic.
9. The system must utilize less than 117k LUTs in the programmable logic.

The system also has a number of trade-off requirements. These requirements are not necessary for the system to function correctly, but it would be beneficial if they are satisfied. They can be used to justify certain decisions:

1. The system should provide a decrease in latency when compared to the algorithms run on the microprocessor only.
2. The system should be modular and easily extendable to conform to various application-driven needs, given the unique constraints of different platforms.

3

Theoretical Background

In this chapter, we will provide the necessary theoretical background to motivate design choices. Before delving into model-specific algorithmic details, we will provide a more general overview of machine learning.

3.1. Introduction to Machine Learning

Given the hardware-focus of this thesis, we will only provide a minimal overview of the broader ranging machine learning context from which our work takes inspiration. We refer the interested reader to the complementary thesis developed by the Signal Processing and Algorithms subgroup.

In traditional, supervised machine learning approaches, the aim is to train a model on a labeled dataset to unravel the input-output relations governing the pre-defined classes of data. A sufficiently well-trained network can ideally generalise these relations beyond its training data and classify entirely new data from those same classes. This concept is illustrated in Figure 3.1.

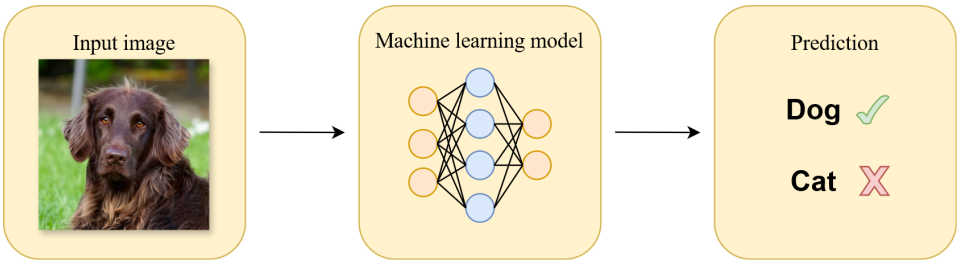


Figure 3.1: General image classification structure. The image is processed by the machine learning network which generates a prediction.

Various machine learning architectures exist, from the classical Multi-Layer Perceptron (MLP) networks [3] or the Recurrent Neural Network [4] to more complex aggregations of networks employed in modern-day machine learning architectures, such as the infamous Transformer [5] which forms the backbone of Large Language Models such as ChatGPT. Essentially, machine learning architectures consist of cascaded layers linearly combining their input and passing it through a non-linear 'activation function,' until a final output is obtained and saturated to generate predictions. We will shortly delve into one architecture in particular, the Convolutional Neural Network (CNN) [6], which comprises the most important part of our machine learning structure.

Importantly, to train a network, machine learning employs the **backpropagation** algorithm [7], a dynamic programme that can numerically evaluate the gradient between any two variables (or nodes) in a directed, acyclic computational graph [8], exploiting the distributivity of the chain rule. The model iterates through the labeled examples and evaluates its performance with a loss function that quanti-

fies the distance between the network's output and the ground truth corresponding to the input. These errors are 'backpropagated' to the model parameters (usually the weights of the linear combinations) to find the gradient of the loss function with respect to these parameters, after which we update these parameters by taking a small 'step' into the direction of steepest descent. This process is repeated until the model converges to a local minimum, which is evaluated against a separate test-set to which the model had no access during training. In practice, this process is somewhat more nuanced, as care must be taken to prevent 'overfitting,' evaluate 'underfitting,' or to select appropriate 'hyperparameters' such as the learning rate which determines the size of the learning step. For a more detailed explanation, we again refer to the Signal Processing and Algorithms thesis or the literature, as the algorithmic specificities are less relevant to this thesis.

3.1.1. Self-Supervised Learning

Traditional machine learning approaches require huge, representative, and, most notably, *labeled* datasets. However, as mentioned above, we are bearing witness to an exponential increase of *unlabeled* datastreams and compute at the edge, which cannot be exploited in the classical framework. Hence, researchers took to developing *self-supervised* models which can train a network on unlabeled data. Backpropagation, however, requires some 'ground truth' to construct the loss function, used to update the model parameters. Various solutions have been proposed to construct a self-supervised loss function anyhow, a particularly salient example being the *non-contrastive learning* approach as employed by, e.g., [1] and [9]. These approaches randomly *augment* the incoming data in various ways that preserve the semantically relevant information; in the case of image data, for example, augmentations might include blurring and cropping. The outputs of the networks are interpreted as 'representations' – vectors embedding the semantic context of the input data – instead of classifications. The loss function compares the outputs of the network for the different augmentations, and punishes very different representations for augmentations of the same image by assigning a high loss to those outputs. Though the outputs of a 'trained' network are by no means meaningful by themselves, downstream classification tasks can exploit this pre-trained encoder to accelerate learning with minimal labeled data, as illustrated by Figure 3.2, potentially achieving high validation accuracies even by only linearly separating representations.

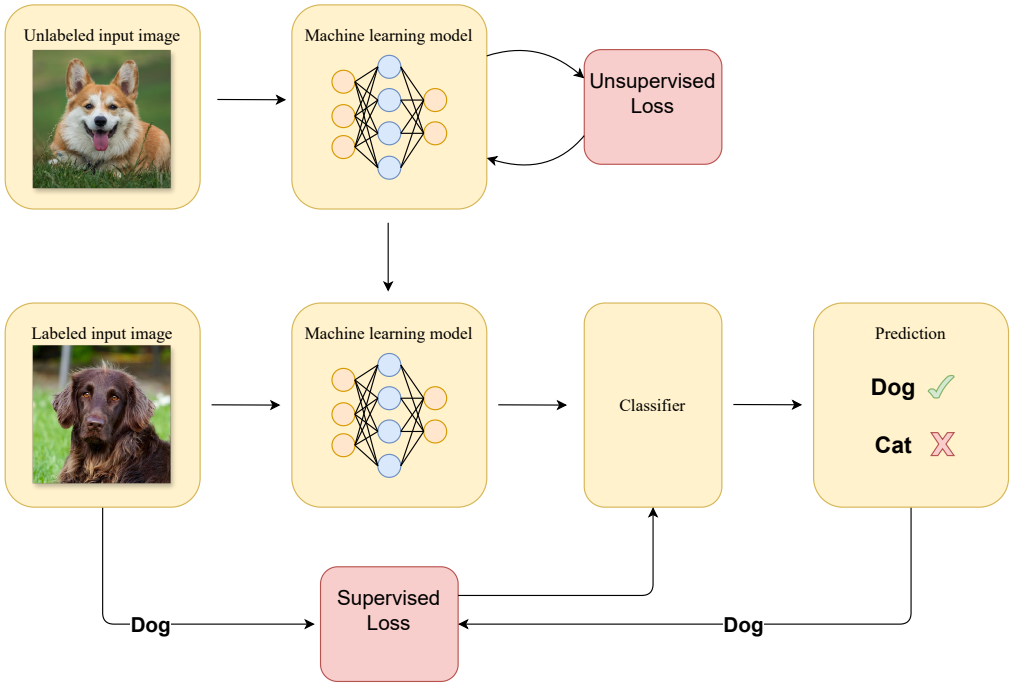


Figure 3.2: General self-supervised learning setup. The self-supervised learning setup trains an encoder using unlabeled data.

These properties make self-supervised learning a promising framework to make use of unlabeled datastreams, and an almost perfect candidate for our move to the edge, if not for its intense computational requirements.

3.1.2. Federated Learning

To fully enable the effective move of self-supervised learning models to the unlabeled datastreams themselves, and to unlock additional dimensions of privacy and bandwidth savings, the recent paradigm of federated learning [10] holds great potential. Instead of centralising machine learning tasks in ever growing Cloud compute-clusters, federated learning attempts to distribute training and inference across multiple, individually less powerful nodes, or '*clients*.' Instead of sending raw data collected by these clients to some central server to train a global model, clients train a local model instead and only share proposed model parameter updates with a central server, which aggregates these propositions and calculates global model parameter updates. The globally updated model is re-distributed among clients to continue local training and inference. The sole dependence on model parameters preserves client-side privacy in case of sensitive information, and in the case of limited on-site resources additionally minimises data communication overhead, proving salient for our purposes by addressing the computational intensity of self-supervised approaches and enabling the use of otherwise inaccessible privacy-protected data. Since this thesis focuses on the development of the client-side architectures themselves, however, we will delve no further into the specifics of federated learning.

3.1.3. The Convolutional Neural Network

Though self-supervised models can be used to train encoders on any data-format, including time-series data such as biomedical signals or audio recordings, our thesis focuses on image encoding in particular. For the purposes of image processing by machine learning architectures, the Convolutional Neural Network (CNN) [6] has proven to be indispensable, and comprises the key component of the employed architecture. The CNN operates by training *kernel-weights*, which represent the coefficients of a small (usually 3×3-sized) filter, rather than entire weight matrices that linearly combine every input to every output. This kernel is convolved with the input image, which occurs by 'sliding' the weighted kernel over the image, multiplying the kernel weights with the underlying pixel-activations, and adding these together, possibly adding a bias-term afterwards. These *local* linear combinations are then fed to a non-linear activation function. As such, the CNN respects the 2D-structure of image data – which we would otherwise lose if pixel-activations were to be fed to an MLP network as a collapsed vector – and uses significantly less parameters. The kernel-operation can be expressed by the following equation:

$$y = w_0 + \sum_{i=1}^{9} x_i \cdot w_i \tag{3.1}$$

Where y represents the output of the kernel, w_0 unto w_9 the parameters of the kernel (where w_0 is a bias-term independent of the pixel-activations), and x_1 unto x_9 the values of the underlying pixel-grid. This output is passed through a non-linear activation function, such as the ReLU function which only admits positive outputs and annuls negative values, which produces a new 'pixel' in the intermediate feature-map of the following layer. This process is repeated until, at some point, the output is flattened into a single vector that may be fed to an MLP network to perform classification.

CNNs typically consist of multiple layers, and any given layer may also consist of multiple *channels*. Analogous to the RGB-channels of a picture, which effectively represent three different 'feature-maps' of the input image, intermediate layers may expand activations across multiple intermediate channels. The value of a pixel in an output channel is calculated by accumulating the results of independent kernels performing their operation on their unique input channel, as displayed by Figure 3.3. Note that the x-grid imposed over the underlying image of a dog is not representative but merely illustrative, as each x-input comprises only a single pixel of a single channel in practice.

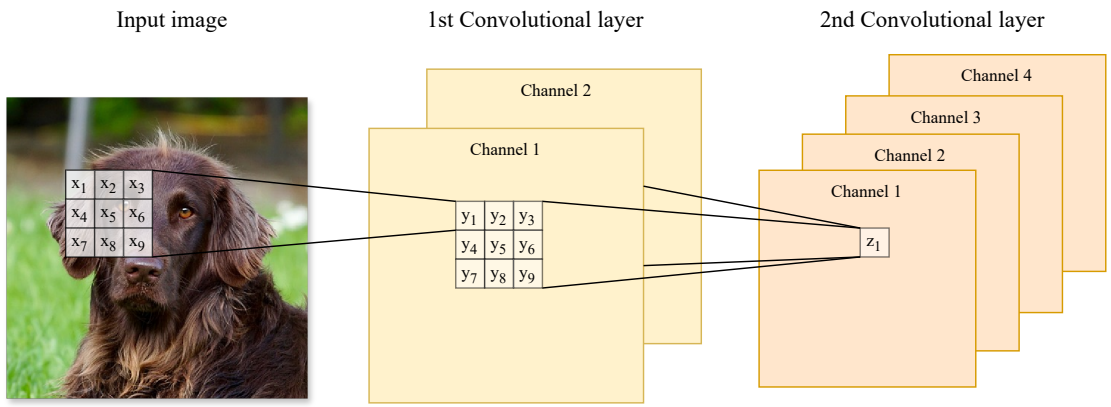


Figure 3.3: Illustrative example of a 2-layer CNN, with the first hidden layer consisting of 2 channels and the second hidden layer consisting of 4.

From a computational point of view, the fundamental computation underlying the operation of the CNN is the aforementioned kernel-operation, essentially representing *multiplication* and successive *accumulation*. Since only the x-values change whereas the w-parameters remain constant over an entire channel, this operation admits both parallelisation and more efficient implementations than the naive, purely sequential approach. Moreover, though CNNs can technically vary herein, our kernel slides over the image with a stride of one – i.e., the kernel moves a single pixel to the right to calculate the next output. This means that only three out of nine inputs change per step, indicating significant memory-reuse.

3.2. Model Description

As the federated learning algorithm bears little relevance to the on-chip implementation, we will only provide a brief overview of the specific self-supervised learning algorithms used.

3.2.1. Self-Supervised Learning Models

This project is primarily inspired by [2], which proposes a *Self-supervised On-device Federated Learning* algorithm (dubbed *SOFed*). The original authors take inspiration from *Bootstrap Your Own Latent*, or *BYOL* [1], to implement the on-device self-supervised training algorithm, and adapt it slightly to make better use of minimal on-device memory resources. This adaptation makes use of a local memory buffer with limited storage capacity, inspired by the limited memory of edge-devices in practice. A problem arises, however, when this assumption is coupled with the large datastreams that may be generated, which the authors address by implementing an *importance scoring* metric. This metric quantifies the novelty of incoming data by calculating its similarity with the presently stored data in the buffer, to make sure the buffer contains a maximally surprising subset of the input data. However, such a metric would only be useful if the speed and volume of incoming data would exceed the processing speed of the device, which we regard as an application-driven requirement. Moreover, FPGAs generally include extendable storage (such as SD-card slots or external SSD ports), which can be scaled-up to meet requirements. Hence, we do not implement the importance scoring metric ourselves.

After extensive investigation by the Signal Processing and Algorithms subgroup, we have ultimately also chosen to adapt SOFed to the *Simple Siamese* self-supervised learning method, abbreviated to *SimSiam*. SimSiam bears multiple advantages compared to BYOL for the purposes of an on-chip implementation, which will be explained in its respective subsection.

Bootstrap your own Latent

As mentioned, the SOFed paper originally utilised the BYOL algorithm which strives to minimise the loss between two different augmented versions of the same image. The method consists of multiple networks split in two separate branches as displayed in Figure 3.4, usually referred to as an online branch and a target branch. Each image in the batch is augmented to obtain two slightly different

images that are each fed to another branch. The branches share an equivalent architecture for the greatest part, consisting of an encoder followed by a projection. The encoder is constituted by a CNN that condenses and collapses the input to a single vector, followed by a projector in both branches and an additional predictor in the online branch – both standard MLP networks. The model parameters of the encoder and projector of the target branch follow an exponential moving average (EMA) of the model parameters of the respective modules in the online branch, to ensure similarity but prevent 'collapse.' The loss function combines the outputs of both branches and assigns a loss on the basis of their dissimilarity, which is used to update the parameters of the online branch but does not backpropagate through the target branch – hence the 'stop gradient' (s.g. in Figure 3.4).

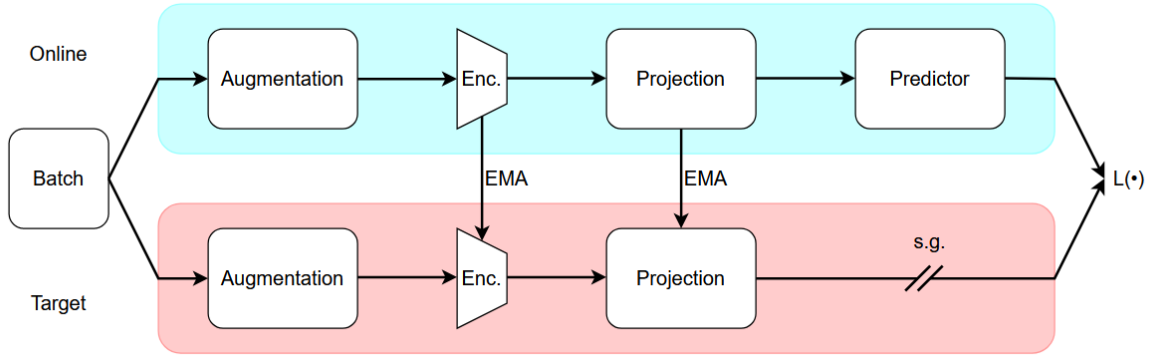


Figure 3.4: Overview of the BYOL self-supervised learning algorithm.

Simple Siamese

The Simple Siamese method (SimSiam) employs a similar non-contrastive strategy as BYOL. However, instead of imposing an EMA on the model parameters of the target branch, SimSiam uses the same exact parameters for both branches, which saves on time and memory requirements. Moreover, SimSiam tends to converge faster than BYOL with smaller batch-sizes, making it an even more attractive candidate for our FPGA implementation. Thus, in consideration of these benefits, we have chosen to commit to SimSiam and abandon the EMA requirement of BYOL.

3.2.2. Datasets

Several datasets have been considered during this project by the Signal Processing and Algorithms subgroup. It was decided to limit the hardware implementation to MNIST (Modified National Institute of Standards and Technology database) [11] and datasets using the same image dimensions and grayscale as MNIST to keep the hardware modules simple. Kuzushiji-MNIST, or KMNIST [12], has been primarily used at the end of the project to see the effects of algorithm modifications and hardware acceleration on the accuracy more clearly. It has the same image parameters as MNIST but is visually somewhat more complex. For a more detailed description of the selection process of a suitable dataset, we refer the reader to the thesis of the Signal Processing and Algorithms subgroup.

4

Identification of Bottlenecks

4.1. Profiling

The most computationally intensive sections of the algorithm were identified by running the algorithms on the microprocessor of the relevant development board and utilising the line-profiler Python module [13] which records the execution time each line in a Python script cumulatively requires. The exact model on which this profiler was run, was a BYOL model with encoders consisting of two layers with respectively 32 and 64 channels. The applied augmentations consisted of a randomly applied Gaussian blur, a random rotation and a random resized crop. The MNIST dataset was used and was divided into batches of 32 images. Federated learning was not utilised during the profiling because it would only differ from the self-supervised learning algorithm by the presence of data transfer between the client and server which cannot be accelerated on the client-architectures themselves.

The execution time of the whole algorithm was mainly spent on training the BYOL model, 96.1% was spent on this section of the algorithm with the rest of time being taken up by the classifier, calculations of statistics and other smaller tasks. This corresponds to an approximate 43 minutes of execution time per epoch. Figure 4.1 shows the distribution of the execution time per section of the BYOL training algorithm. It can be seen that the most intensive sections are the augmentations, encoders and the backpropagation (likely primarily across the encoders), respectively requiring 18.4%, 34.5% and 40.3% of the total execution time. It should be noted that these sections mostly use parallel computations which could benefit greatly from dedicated acceleration.

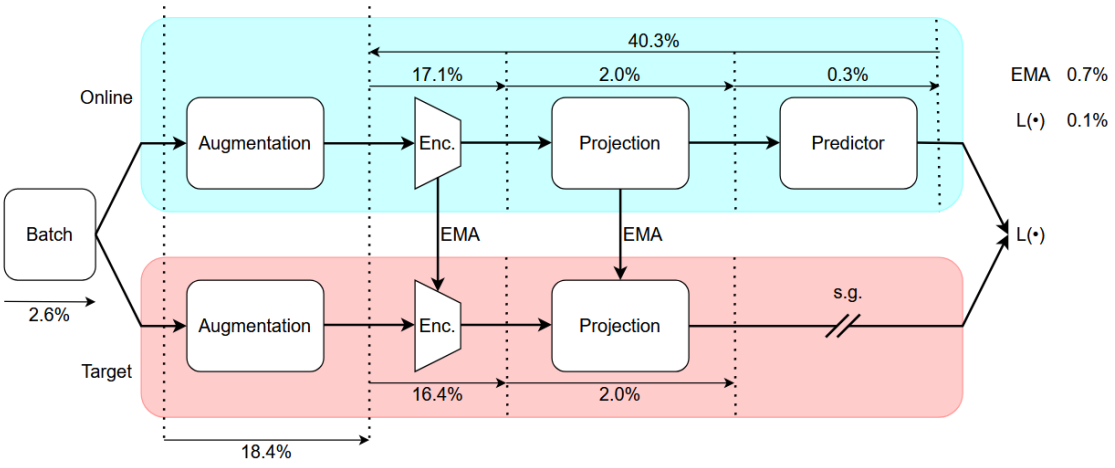


Figure 4.1: Relative execution time of the different modules within the BYOL training algorithm as obtained by using the code profiler.

Based on these profiling results, a number of functionalities were selected to be implemented as specific modules within the programmable logic to accelerate this training process. These modules include a Convolutional Layer Accelerator which can be utilised to replicate the CNN in the encoders and to apply the Gaussian blur augmentation. Two additional augmentation modules have been designed to also perform the rotation and resized crop augmentations. Despite the massive contribution the backpropagation has to the total execution time, its implementation would require a successful forward-pass architecture in the first place, due to which we have determined that it falls outside of the scope of this project.

4.2. Theoretical Motivation and Design Space Exploration

When investigating the adoption of dedicated hardware acceleration, it is important to identify the primary computational bottlenecks of the original algorithm, as we have done in this chapter. Once the computational intensity of key algorithmic blocks has been profiled – e.g., as we have done in terms of total and relative CPU time – further analysis into the specific causes of the bottleneck is warranted. After all, algorithmic blocks may experience high latency for a multitude of reasons, not all amenable to acceleration. Most significantly, in the move to programmable logic, fully *compute-bound* algorithms can enjoy a great deal of acceleration whereas fully *memory-bound* algorithms cannot be accelerated at all. In the case of machine learning, the classical feed-forward MLP depends on huge weight matrices that store the *unique* weight between every input node to every output node per layer. This massive memory-dependence leaves little room for improvement and thus allows little to no acceleration. As no parameters can be reused, in contrast to the CNN, the read-out of the requisite model parameters will likely take longer than the fairly trivial calculation. Conversely, CNNs show a high degree of memory reuse and can benefit immensely from parallelisation.

Luckily, the profiling results align nicely with the amenability to acceleration. An accelerated image augmentation module essentially entails a dedicated image-processor that can perform certain essential augmentations, as we shall delineate in the next chapter. Most augmentations, such as blurring or rotation, can be performed in an extremely parallel fashion, incurring an area-latency trade-off similar to the CNN. In the following chapter, we will describe the hardware implementation of these modules. Normally, design of hardware modules would warrant a preemptive design space exploration, identifying the constraints of the available on-chip resources to determine an appropriate architecture. However, in our approach, we have embraced a design philosophy of easy and modular extensibility. Every module consists of key algorithmic accelerators that can be instantiated in parallel if latency remains a problem, slightly complicating control logic and at the expense of area-use. This design philosophy is also in line with the spirit of this thesis, developing a general framework and key accelerators that can be modified to fit specific application-driven needs. Finally, the successive augmentation-encoding can be implemented as a streamlined pipeline in the programmable logic, producing an efficient and extensible baseline architecture.

5

Hardware Acceleration

5.1. Convolutional Layer Accelerator

Perhaps the foundational element of the dedicated accelerator framework is the Convolutional Layer Accelerator, or the CLA. The CLA is responsible for accelerating the *multiplication* and *accumulation* of the convolution-operation, as well as performing the subsequent activation function. As mentioned in subsection 3.1.3, as the convolutional kernel slides over the image, we observe a high degree of memory reuse. Moreover, spatial convolution is fully parallelisable; in theory, all convolutions over the entire image could take place at once, as there is no interdependence between different outputs. However, such a high degree of parallelism could no longer exploit the memory reuse, and would incur extraneous area use. Hence, in line with the design philosophy set forth in the previous chapter, we have developed an easily duplicable filter-unit, which consists of its own *pixel buffers* and *Multiply-and-Accumulate* unit, henceforth *MACC*-unit. The MACC-unit concurrently performs the activation function, exploiting the inherent non-linearity of quantization and saturation. These two units are controlled by an overarching Finite State Machine (FSM) that oversees the control logic underpinning the filter-unit. We will now describe the specific construction of the filter-unit by describing its functional components.

5.1.1. The Pixel Buffer

To exploit the high memory reuse inherent to convolution, we construct *pixel buffers* that obtain and store local image data in on-chip registers. Inspired by [14], each pixel buffer buffers a single *row* of the input image in its entirety. Since every filter-unit is equipped with a single MACC-unit (though, in theory, could employ multiple MACC-units at once), and since the kernels in our design are strictly 3×3-sized, we need at least three pixel buffers to make sure the right data is buffered in the registers. This would also allow the filter-unit to exploit the high memory reuse, as three rows of an image of width n produce the entire first $n-2$-wide row of the output layer, i.e., they support $n-2$ convolutions. Moreover, any two consecutive convolutions reuse six of the same input pixels, and the three new pixels can be obtained by incrementing the read-addresses of all three buffers by one at once. Once a kernel arrives at the end of the row, it moves a single row downward to continue its operation – in other words, the second and third pixel buffers of the previous row-convolution comprise the first and second buffer of the new row-convolution respectively.

To fully exploit this level of row-based memory reuse, we instantiate a *fourth* pixel buffer which loads an entirely new row into its registers whilst the first three rows are actively being read to perform the convolutions. This way, the MACC-unit can seamlessly continue operation across rows. To illustrate the level of memory reuse, Figure 5.1 displays three separate convolutions and their respective output pixel across four rows, as well as their shared input data.

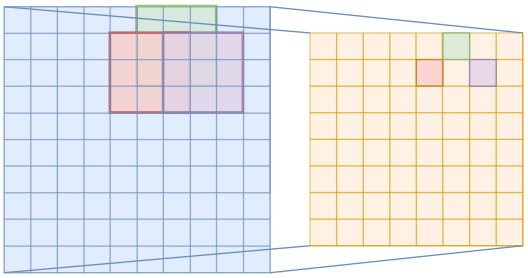


Figure 5.1: Illustrating the memory reuse exhibited by four-row buffering.

To read the pixel buffers, an address must be supplied, corresponding with the number of the pixel in the given row, which returns a 24-bit wide vector representing the pixel value at the given address as well as the values of the two immediately following pixels. To make sure the right pixel-values are fed to the MACC-unit, we must implement some control logic to multiplex the buffer outputs, concurrently ensuring that one buffer is kept aside to load in new data while the MACC-unit processes the preceding 3 rows. A high level overview of the buffer-multiplexing logic is displayed in Figure 5.2.

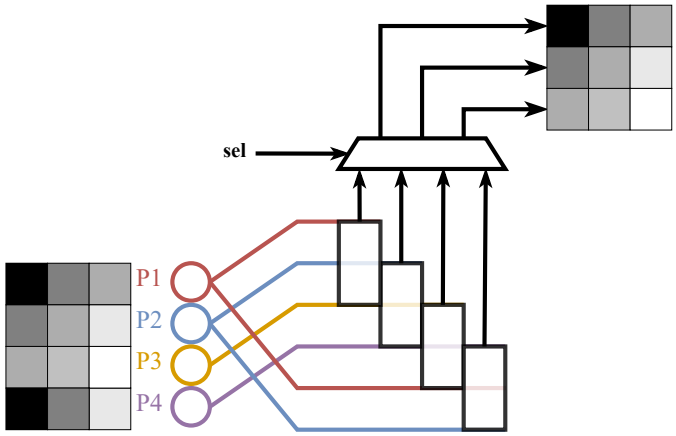


Figure 5.2: Multiplexing the outputs of the pixel buffers to ensure the right information is fed to the MACC-unit

The ‘read_address’ input of the pixel buffers is controlled by an FSM that keeps internal count of the number of ‘convolved’ output-pixels generated for a given row in the output channel. Once an entire row has been generated, the ‘read_row_counter’ is incremented as well and ‘read_address’ is reset, which adapts the ‘active_read_state’. This signal functions as the selection bit (‘**sel**’) of the multiplexer in Figure 5.2, concurrently making sure that the ‘read_address’ signal is fed to the correct pixel buffers. At all times, the pixel buffers output a vector containing 3 pixels, stored at addresses determined by read_address unto read_address+2.

5.1.2. Multiply-and-Accumulate Unit

The module actually performing the kernel-operation is the multiply-and-accumulate unit – i.e., the MACC-unit. The top-level FSM of the filter-unit collects and provides a 3×3-sized pixel-grid as well as the 3×3-sized kernel; that is, a unit external to the filter-unit reads out and actively buffers the kernel’s weights such that they are accessibly at all times, which the filter-unit internally connects to the MACC-unit. The MACC-unit has been pipelined to minimise latency. First, the unit multiplies the kernel weights with their respective pixel-value, and stores these in an intermediate register. These register values are fed to an accumulator, which loops through the nine outputs and adds them, embedding a recursive adder tree. This value is shifted to the right by a fixed number of bits to allow for decimal-level precision of the weights, and is then passed through an activation function to generate an output ‘pixel.’

To save on memory and communication overhead and to speed-up operations, we quantize the input,

kernel weight, and output values. For the input layer, which is fed an augmented image, we interpret the values as unsigned 8-bit integers, which fully captures the original data as the majority of image-data can be represented by purely positive 8-bit integers. For the weights, we have opted for 12-bit quantization, as the Signal Processing and Algorithms group has experimented with various quantization formats and has found that 12-bit weight quantization – as done in the hardware – achieves satisfactory results. The MACC-unit performs signed multiplication between the (signed) weights and inputs, which, after accumulation, is shifted to the right by 7 bits – which is equivalent to a division by 128 in decimal notation or, similarly, the interpretation of the 12-bit weight as two's complement fixed-point representation comprising 1 sign bit, 4 integer bits, and 7 fraction bits. This means that, effectively, the inputs are multiplied by weights in the range of $[-16,\ 15.9921875]$ as the range of two's complement fixed-point numbers is given by $[-2^{N-1} \cdot F,\ (2^{N-1}-1) \cdot F]$ where N is the total number of bits and $F = 2^{-f}$, f being the number of fraction bits. This range consists of 4095 unique, equidistant points, with a distance of 2^{-7} or $\frac{1}{128}$ between any two values. Since the inputs are interpreted as 8-bit unsigned integers (but, for intermediate layers as 8-bit *signed* integers, since the sign may appear after batch normalisation), the outputs of the MACC-unit are constrained by $[-4080,\ 4078.0078125]$ (or, respectively, $[-2032, 2031.0078125]$). Of course, this exceeds the range of the 8-bit unsigned integer output – which is addressed, simply, by clipping the output between 0 and 255. That is, negative outputs are set to 0 whereas outputs exceeding 255 are saturated to 255.

Since we clip the output, we effectively transform the data in such a way that this already embeds the non-linear activation function. Algebraically, the clipping embeds the function:

$$y = min(max(0,\ x),\ 255) \tag{5.1}$$

Which bears resemblance to a shifted, first-order approximation of the popular *sigmoid* activation function. This function has been observed and used before, dubbed across literature as the 'hard sigmoid' or 'ReLU-6' activation function [15] (though the linear region of the hard sigmoid usually spans $[0,\ 1]$ and 'ReLU-6' clips fixed-point outputs between 0 and 6 instead). Hence, we can exploit the inherent non-linearity of clipping the MACC-data to comply with the 8-bit output as the activation function of the CNN, such that we do not need to design an additional unit performing the transformation itself.

5.1.3. Filter-unit Control

Finally, these units need to be controlled by overarching control logic, as alluded to above. The filter-unit employs two distinct FSMs: one FSM waits until three pixel buffers have been loaded in completely before starting convolution, and contains an additional state to stall reading-out the buffers in case the memory bandwidth cannot match the speed of the operations, and the other FSM actually steers the MACC-unit and can similarly stall convolution for the same reason. The filter-unit, moreover, also embeds the combinational multiplexing-circuit of Figure 5.2, to ensure pixel buffers are read out in the right order.

5.2. Image Augmentation

When running the algorithms solely on the microprocessor, the image augmentations are performed by using the Python library Torchvision [16]. Both the BYOL and SimSiam training models designed by the Signal Processing and Algorithms subgroup use three specific augmentation types that include several randomised variables to obtain a wide range of possible input images. These augmentations include: randomly applied Gaussian blur, rotation, and resized cropping. These augmentations have been implemented in a simplified manner on the hardware.

5.2.1. Randomised Augmentations

The performed augmentations using Torchvision are randomised to ensure the model can distinguish between different groups of images, even if small variations are introduced. Several aspects of the augmentation process can be randomised allowing a great variation between the images entering the encoders. In the Python implementation, each image has a 20% chance to be modified using Gaussian blur and several augmentation variables are selected within a certain range, including the Gaussian kernel, the degree of rotation, the cropping scale and the location of the crop.

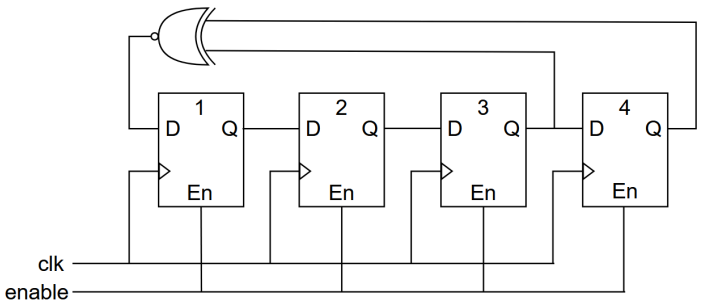


Figure 5.3: Architecture of a 4-bit LFSR

It is not possible to cover all these ranges in an entirely random manner within hardware. Instead, for each variable a number of fixed values was chosen and one of these values is pseudo-randomly selected for each new image. This decrease in possible values was made in order to simplify the implementation of the augmentations to allow for Look-Up Tables (LUTs) which would only require one clock cycle to compute the output pixels instead of a number of cycles as would be the case when performing parameterised calculations. This results in easier processing of the images and an architecture which can process images pixel by pixel. This method requires images to be completely stored within a memory module before each LUT-based augmentation to successfully map them to their augmented version.

For the selection of the variable values, it was decided to use Linear-Feedback Shift Registers (LFSRs) to create pseudo-random number sequences. An LFSR is a shift register whose input is determined by an XNOR operation performed on a number of flipflop outputs within this register, as is illustrated in Figure 5.3 [17]. Normally, the flipflop outputs are updated every clock cycle but an 'enable' signal was added so this only occurs at the start of each image. The number sequences generated by these modules have a fixed period, but it has been assumed that the combination of different images and different LFSR periods for different random variables provides enough random image representations to simulate the desired effect. Additionally, the LFSRs have been designed to have the maximum period duration possible given their respective sizes.

5.2.2. Gaussian Blur

A Gaussian blur augmentation is normally implemented by performing a convolution on the input image using a Gaussian kernel. The Python-implemented Gaussian blur is applied to 20% of the input images using a 3×3 kernel whose exact values are determined by its standard deviation which ranges from 1.0 to 2.0. On-chip, the decision to apply a Gaussian blur is determined by a 2-bit output of a 9-bit LFSR. Three of the possible combinations will result in no Gaussian blur being applied while the fourth combination will result in a Gaussian blurred image. Thus, each image has a 25% chance to obtain a Gaussian blur of standard deviation 1.0 when the hardware module is used.

In Section 5.1, the design and implementation of the CLA have been examined. The Gaussian blur is effectively a convolution using a pre-determined Gaussian kernel and can be implemented using the same design as discussed above. However, one significant discrepancy occurs between the standard convolution and the Gaussian blur implementation using Torchvision. The output of a convolution has smaller dimensions than the original image when no padding is used as is the case with the discussed CLA design, resultant from the inherit resolution-loss of 2D spatial convolution. Meanwhile, the Gaussian blur should have an output image with the same dimensions as its input. Since the border of images usually contains no relevant data for the MNIST dataset and its variants, we have modified the CLA-architecture to zero-pad the output image with a 1-pixel wide border on all sides. This happens by omitting some addresses when writing the resulting image in an internal memory buffer, leaving their contents to be zero. In case of more elaborate images, the image might need to be zero-padded, or padded in some other way, before the convolution occurs to minimise visual artefacts.

5.2.3. Rotation

The rotate augmentation in Torchvision rotates an input image with a number of degrees around its center [16]. In the Python-based models, the images are rotated by a value randomly chosen from a range between -30 and 30 degrees. In developing a hardware implementation, we investigated the algorithms employed by rotation. If we assign an $(x,\ y)$-coordinate to every pixel centered around the middle of the image, which we have taken to be the 14th pixel to the right and 14th pixel down in case of 28×28-sized images of the MNIST-variations, we can calculate the new position of the pixel in the rotated image by simply multiplying it by the 2D-rotation matrix:

$$\begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{5.2}$$

Where $\hat{x}$ and $\hat{y}$ represent the rotated coordinates. However, since the CLA functions by accepting a continuous stream of *consecutive* pixels, we have adapted Equation 5.2 to express the original coordinates in terms of the rotated coordinates. This expression requires the inverse of the rotation matrix, which equals its transpose and exists for all θ:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} \tag{5.3}$$

This allows us to find the original pixel coordinate corresponding to any coordinate in the desired rotated image. Of course, on-chip, the pixels of 28×28-sized image are stored by a vector with indices ranging from 0 to 783. Luckily, this conversion is quite simple:

$$\text{index} = (x \ \%\ 28) + (y\ //\ 28) \tag{5.4}$$

Where % is the modulo-operator and $//$ floored division. To translate these insights to an on-chip implementation, we could implement arithmetic modules that can calculate the address given the index of interest and some angle of rotation θ in real-time, but we have opted for pre-compiling these index values for four set rotations: -30, -15, 15 and 30 degrees. We store these indices on a LUT to which we attach a counter which can iterate through all indices. This counter returns the addresses in the order of the output pixels, which are fed to the memory to retrieve the appropriate pixel value for said 'new' pixel in the rotated output image. The number of different variations has been kept small because an increase of possibilities would also require equally more LUTs. To simulate the randomised selection of an angle of rotation, we attach the outputs of the rotation module (corresponding to the angles of rotation) to a 2-bit multiplexer, which is controlled by an 11-bit LFSR.

5.2.4. Resized Crop

The last implemented augmentation is referred to as random resized crop within the Torchvision library [16]. A smaller square section of the input image is taken and this cropped image is then resized to its original dimensions. The size of this cropped image in Python is a fraction between 0.5 and 1 of the original image size corresponding to a height and width between 14 and 28 pixels for the MNIST-like datasets. The location of this crop within the original image also varies when using Torchvision. However, it was decided to limit the location to the center of the image to simplify the implementation. The size of the cropped image before resizing is determined by a 10-bit LFSR and includes dimensions of 16, 20, 24, and 28 pixels.

After cropping, the image should be resized to the original image size of 28 pixels. Torchvision uses bilinear interpolation in which the value of a pixel is calculated by taking the weighted average of the pixels surrounding it. This interpolation method was considered but it was decided to use the nearest neighbour interpolation method instead. This method determines the value of the output pixel by calculating which pixel of the cropped image is closest by and copies its value. This requires less computations then bilinear interpolation and is more aligned with the general design of the hardware acceleration which assumes pixels enter and leave the modules sequentially. When using nearest neighbour, the mapping of the pixels after resizing can be priorly determined for the different scaling possibilities, again, allowing the use of LUTs in a similar fashion as the rotation module. This greatly simplifies the necessary computations and enables the incoming pixels from the data stream to be directly mapped to their new locations.

5.2.5. Additional Augmentations

An additional, salient feature of the CLA-architecture is its wide applicability as the generator of many different augmentations. Much like it has been used to generate the Gaussian blur as described above, it could implement any spatial convolution that uses a 3×3-sized filter. These could include horizontal or vertical Sobel filtering, sharpening kernels, or other such visual augmentations. This easily configurable property is in line with the aforementioned design philosophy of extensibility and adaptability.

6

Overview Hardware Set-up

6.1. KRIA KV260

The client node of the federated learning set-up and the FPGA embedding the hardware acceleration is implemented on a KRIA KV260 System-on-Module [18], henceforth referred to as the KRIA. The KRIA contains a ZYNQ Ultrascale+ Microprocessor System on Chip (MPSoC) [19] which runs a GNU/Linux Ubuntu 22.04 Operating System and can be used to run the standard algorithms on Python. Additionally, it also includes programmable logic (PL) which can be used to implement the hardware acceleration modules. The Ubuntu Operating System enables simplified access and control over the functionalities of the board. A connection can be made between an external computer and the KRIA over Secure Shell (SSH) or Ethernet as long as the devices are connected to the same network. This connection will also be utilised to implement the communication between the server and the client node essential for the federated learning algorithm. The PL is programmed by using the Vivado Design Suite [20] and is operated on its standard clock frequency of 100 MHz. A representation of the ZYNQ Ultrascale+ MPSoC can be connected to custom hardware or a selection of Intellectual Property (IP) cores using the Advanced eXtensible Interface (AXI), an interface protocol used by nearly all IP cores within Vivado and the Zynq Ultrascale+ MPsoC.

6.2. Communication

6.2.1. PYNQ

The algorithms implemented by the Signal Processing and Algorithms subgroup have been written in Python and can be run on the microprocessor of the KRIA. PYNQ [21] is an open-source project from Advanced Micro Devices (AMD) that allows an user to access the programmable logic with Python running on their ZYNQ microprocessors. The programmable logic can be configured via Python, and data and control signals can be transferred bidirectionally enabling close communication between the platforms.

6.2.2. Data Transfer

The hardware acceleration modules need data from the processing system to do their computations. The augmentations and the forward-pass of the CNN have been moved to the programmable logic. In order to perform these functions, the microprocessor needs to supply the input images and the model parameters to the programmable logic modules. Additionally, outputs of the hardware modules need to be transferred back to the microprocessor to allow for loss calculation and backpropagation. The weights are quantized and are represented by 12-bit fixed-point values. The images themselves consist of pixels represented by 8-bit integers. In case of MNIST and similar datasets, the image dimensions are 28 by 28 pixels for a total of 784 pixels [11].

Two of the provided IP cores in Vivado are the AXI Block Random Access Memory (BRAM) Controller [22] and Block Memory Generator [23]. The latter acts as the actual memory bank that can be accessed by both the processing system and the programmable logic. Importantly, the read-out speed of the

memory bank in the programmable logic equals *one word per clock cycle*, or, equivalently *32 bits per clock cycle*. The BRAM controller functions as a bridge between the processing system and the memory to ensure the AXI output of the processing system is transformed to be able to interact with the native interface of the memory bank. The custom hardware modules can be directly connected to the ports of the memory. PYNQ has a Mapped Memory Input/Output (MMIO) module that can be used to instantiate BRAM controllers to which data can be written and from which data can be read as long as the data is in the form of 32-bit integers. This method acts as the standard mode of transportation of large quantities of data between the two platforms.

Due to the MMIO module only being able to write and read 32-bit integer values, data with a non-integer datatype is transformed into an integer datatype while keeping its original binary information. Additionally, to maximise the efficient use of memory and decrease the communication overhead, integer values smaller than 32 bits are grouped together. This way, two kernel weights or four pixels can be sent at the same time.

Two additional modules have been implemented to allow this data transfer from the hardware side. A simple read module is able to read the kernel weights stored in a predetermined section of the input memory in order to store it into registers that can be accessed by the relevant CLA modules. This module is also able to read the images and send a continuous stream of pixels to the rest of the hardware modules which can be interrupted at all times by asserting an interrupt signal. The data is written back to microprocessor-accessible memory by using a so-called write module. As long as valid pixels enter the module, they are again combined in groups of four to maximise memory efficiency.

Another data transfer method called Direct Memory Access (DMA) was briefly considered by using the AXI DMA Controller IP [24] in hardware and allocating a section of the DRAM for use by the IPs in the programmable logic via PYNQ. Using DMA, data can be sent quicker and during execution of other parts of the Python program. There is no need to wait until the last bytes have been written because only a start signal is given. Instead of the traditional AXI, this method uses AXI Stream which enables a data stream instead of memory-mapped data transfer. The chosen system of using the AXI interface enables custom modules to be connected to the native interface of Block Memory Generators while these modules are connected via BRAM controllers to the microprocessor. Connecting the custom modules to the microprocessor by using the AXI stream interface was deemed too complex considering the scope of this project.

6.2.3. Control Signals

The PYNQ library also supports another module that can be used to communicate between the microprocessor and programmable logic: the General Purpose Input/Output (GPIO) module. This module can be used to write and read signals that can be used as control signals. From a hardware point of view, an AXI GPIO IP block is used as a bridge between the processing system and ports of the custom modules. These signals include the external signals from the previously discussed hardware acceleration modules, such as a reset and indications when a process has to start or when it is finished.

The time it takes to transfer a control signal from the microprocessor is many times higher than the scale at which the programmable logic operates, namely 10 nanoseconds per clock cycle when using a system clock of 100 MHz. Due to this communication method taking a large amount of time, it is preferred to keep the hardware control as much as possible contained within the hardware. As a result, the only control signals between the microprocessor and programmable logic would be a system reset, a general start signal and several signals that indicate that part of the process is done and the microprocessor can start reading the relevant data from the memory banks within the programmable logic.

6.3. Overview Design

A high-level overview of the design as given in Figure 6.1 shows that the system can be divided into two sections: the processing system and the programmable logic. The processing system includes the Python interface used to control the programmable logic applications but the main focus of this thesis lies within the programmable logic. The programmable logic design can be used in two different situations as shown in the diagram: augmentations of input images followed by a convolutional layer

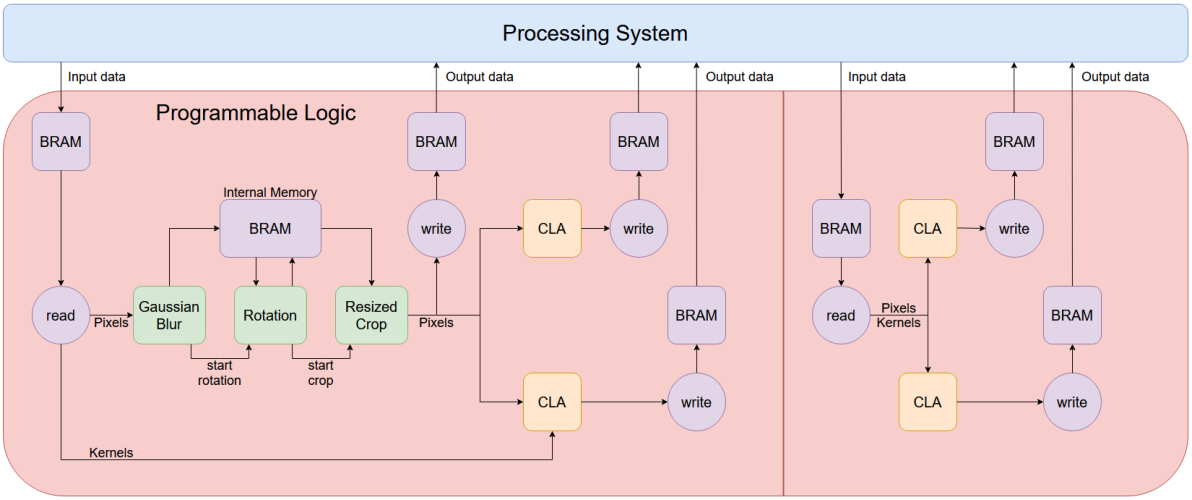


Figure 6.1: High-level overview of the hardware design and its connections to the processing system.

and a convolutional layer by itself.

For both situations, input images and kernel weights enter the programmable logic via a BRAM controller and a Block Memory Generator as discussed in the previous section. The model parameters are written into memory when they are initialised or updated and when the processing system asserts the 'read_kernel' signal, the weights are read using the read module and transferred to their relevant CLA modules. The input images are transferred from the microprocessor to programmable logic per batch. When using the model, two different situations can be applied, an epoch can consist of several batches that are switched out each time or the input memory acts as a buffer which is periodically updated but not totally exchanged. The start signal from the microprocessor indicates when the model should perform its forward propagation. The images are read pixel by pixel by the read module.

When image augmentation is performed, the pixels enter a modified CLA which has a 25% chance of applying a Gaussian blur after which they are briefly stored in an internal storage system by a modified write module as this memory is able to store one byte per address. As discussed in Section 5.2, the rotation and resized crop modules utilise a number of LUTs to directly transform an input image to its augmented version. However, to allow this, the entire input image should already be loaded in. Thus, the images are stored between the Gaussian blur and the rotation, and between the rotation and the crop. To speed up the process, in each transition two images are stored to ensure no images are overwritten. While one of the two image locations is being read, the previous module can write to the other image location.

After all augmentations have been performed, the pixels enter the CLA in which the actual convolutions and activations are performed. Figure 6.1 shows two CLA modules in both situations, corresponding to a two channel layer, but a larger number can be used if so desired. In case of the second application, the pixels enter the CLA immediately after being read from the input memory. The output pixels of the fully augmented images and the results of the CLAs are also transferred to the microprocessor by using write modules and memory banks. The microprocessor receives a signal which is asserted when all images of the batch have passed a certain segment of the process. This way, the microprocessor can start reading the relevant memory to obtain these values without any danger of the operations not being done yet.

Each of these discussed modules can be active at the same time creating a pipeline in which several images are processed at once. This does not decrease the time it takes to process one image but greatly improves the throughput of the entire system.

6.4. Storage Management

The implemented Block Memory Generators need a number of BRAM units. The number of BRAMs which is utilised within this system is dependent on a number of variables and limitations as listed below:

1. The amount of data storage necessary is dependent on the dimensions of the images (N), the number of images processed in series (I) and the number of channels implemented (C).
2. A Block Memory Generator can only be connected to two other modules, one of which often is the BRAM controller connected to the processing system.
3. One BRAM unit is able to store 4 KB of data.
4. The size at which Block Memory Generators can be implemented is limited to a selection consisting of 2^n KB with n an integer equal to or larger than 2.

The last two limitations can be taken into account by using the function shown in (6.1) with as input the number of bytes, equal to the number of pixels in most cases, and as output the number of BRAMs which will be required for this amount of bytes. The number of bytes is translated to number of KB after which the minimum BRAM size is calculated necessary for this amount of data. This number is then converted to the actual number of BRAMs by dividing it by a factor 4.

$$S[x] = 2^{\lceil log_2(x/1024)\rceil - 2} \tag{6.1}$$

Different modules require their own Block Memory Generators due to limitation 2. Their respective number of implemented BRAM units can be calculated by using (6.1). The input memory needs enough storage to contain all input images and to contain the kernel weights for each channel. The internal memory buffer for the augmentations consists of one BRAM due to their fixed size. The memory which stores the intermediary results consisting of the augmented images is dependent on the number of images and their dimensions. The necessary memory storage of the outputs of a CLA module can be evaluated using a similar method but its dimensions is slightly smaller. This number also has to be multiplied by the number of channels as these need their own Block Memory Generator in the current system design.

Equation (6.2) can be used to calculate the total number of BRAM units used when implementing the application that combines augmentation and the first convolutional layer. When only using the CLA, the equation can be simplified to (6.3). The programmable logic of the KRIA has 144 BRAM units available. According to the derived formula and this limitation, the number of images that can be processed in series is for example 167 when assuming images of 28×28 pixels and only one implemented channel of the CNN.

$$B = S[N * N * I + 20 * C] + 1 + S[N * N * I] + C * S[(N-2) * (N-2) * I] \tag{6.2}$$

$$B = S[N * N * I + 20 * C] + C * S[(N-2) * (N-2) * I] \tag{6.3}$$

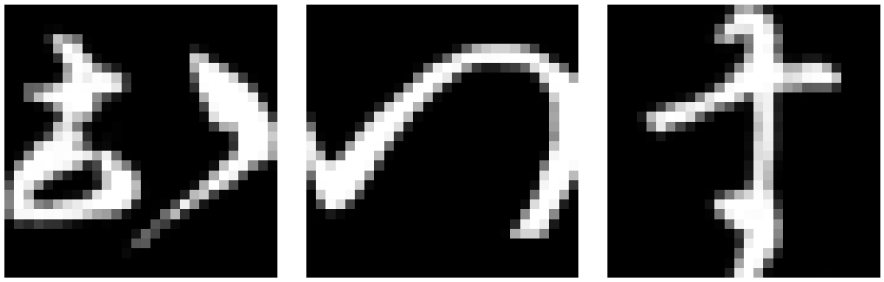
7

Results

7.1. Individual Modules

The hardware modules designed in Section 5 are meant to replace the functionality of already existing functions in the Torch and Torchvision libraries. The performance of the hardware modules is analysed by performing the augmentations on the same image of the KMNIST dataset [12] with the same variables using the hardware modules and the Python functions and visually compare the two cases. The images that have been used in these tests are shown in Figure 7.1.

The functions in Python are done on floating-point values due to limitations of the libraries and to ensure resemblance to the method which is used in the Python-only version of the models. The hardware modules assume the images consist of 8-bit integer pixels. To allow for comparison, the float values have been transformed to equivalent integer values. The integer values range from 0 to 255 and are visually represented by gray-scale pixels within the shown images.



(a) Convolution and Gaussian blur **(b)** Rotation **(c)** Resized crop

Figure 7.1: Original KMNIST images used to display the effects of different augmentations.

7.1.1. Convolutional Layer Accelerator

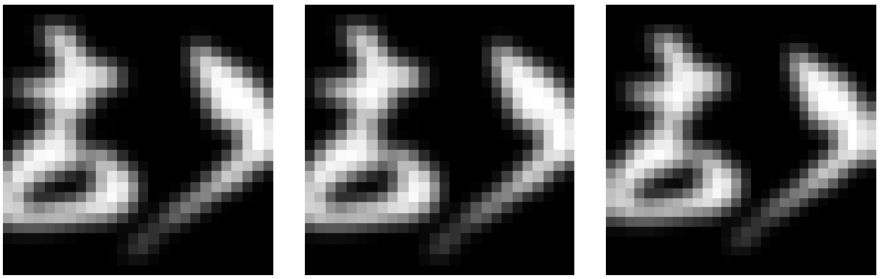
The Convolutional Layer Accelerator has been tested by loading a Gaussian kernel and an image in the memory buffer after which the convolution is performed and the resulting image is written back to memory. A single CLA is fed one pixel per clock cycle, which complies to its operation. In this way, it takes 28 clock cycles to fill a pixel buffer with an entire row, and 84 clock cycles to fill three row buffers such that the MACC-unit can start operation. However, since the MACC-unit takes 25 clock cycles ($width - kernel\ size$) to traverse three rows entirely, with an input bandwidth of one pixel per clock cycle, the convolutions have to be stalled for three clock cycles – i.e., the CLA becomes *memory-bound*. In theory, the CLA could function with higher input speeds such as 2 pixels per clock cycle, though this means it would take 14 clock cycles for a new pixel buffer to fully load-in a new row. Hence, the CLA would become *compute-bound* again, though a single filter-unit could employ multiple

MACC-units concurrently to overcome this limitation.

Either way, to simplify control logic and for the purposes of a proof-of-concept system, we limit the input bandwidth to one pixel per clock cycle.

The image shown in Figure 7.2a is an example of a convolutional forward-pass on hardware using the 3×3 Gaussian filter, thus also representing the output of the Gaussian blur augmentation-unit. While the Torch library provides a two-dimensional convolution function, the Torchvision library also provides a function which can directly apply a Gaussian blur. Both functions were used in the models of the Signal Processing and Algorithms subgroup, so both were analysed and compared with the hardware application. The Torch convolution (Figure 7.2b) is done using a manually declared kernel equal to the one used in the hardware, while the Torchvision Gaussian blur (Figure 7.2c) uses a pre-defined kernel with a standard deviation of 1.0.

The Gaussian blur function returns a 28×28 image while the other two images are only 26 pixels in both dimensions. The border of the Gaussian blur image has been removed to correctly calculate the differences between the hardware and Python results. These differences are shown in Figures 7.8a and 7.8b.

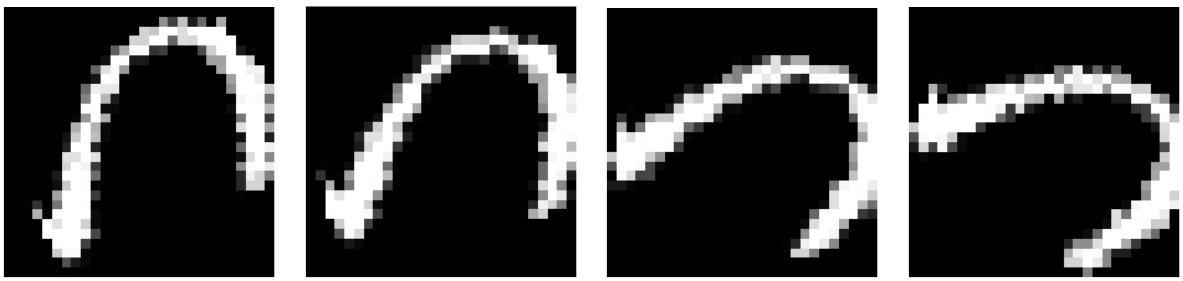


(a) Hardware convolution **(b)** Torch convolution **(c)** Torchvision Gaussian blur

Figure 7.2: Resulting image when a hardware convolution, Torch convolution or Torchvision Gaussian Blur has been applied.

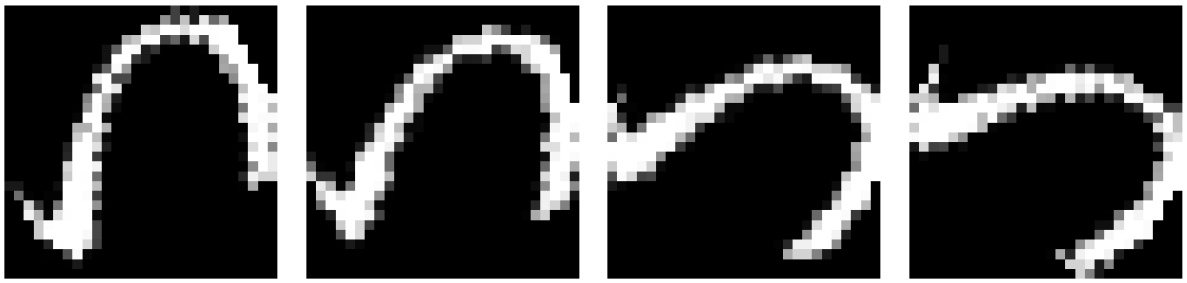
7.1.2. Rotation

The rotation hardware module performs one of four rotations on the input image: 30, 15, -15 or -30 degrees. Images are loaded into a memory buffer via the Python interface after which they are read by the programmable logic, rotated and written back into memory so the augmented image can be read by the interface. The hardware outputs corresponding to the four variations are shown in Figure 7.3. The Python results have been obtained by executing the Torchvision rotate function on the same image, varying the number of degrees, and are shown in Figure 7.4. The absolute difference in pixel values between the resulting images of the 15 degrees rotation is shown in Figure 7.8c.



(a) 30 degrees rotation **(b)** 15 degrees rotation **(c)** -15 degrees rotation **(d)** -30 degrees rotation

Figure 7.3: Image being rotated in all four possible positions using the hardware module.



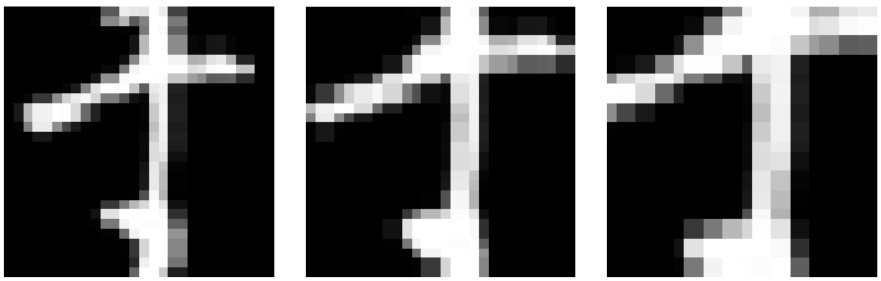
(a) 30 degrees rotation **(b)** 15 degrees rotation **(c)** -15 degrees rotation **(d)** -30 degrees rotation

Figure 7.4: Image being rotated in all four possible positions using the Torchvision function.

7.1.3. Resized Crop

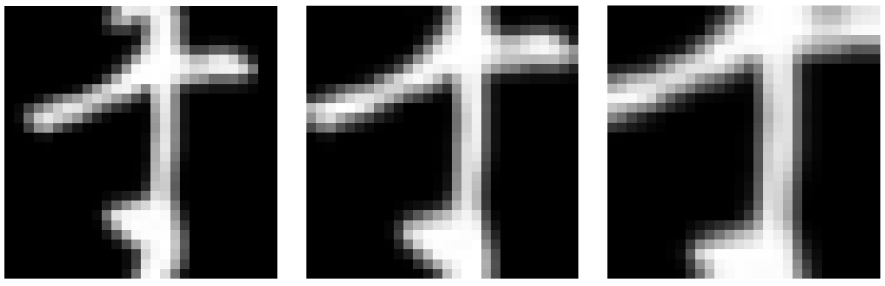
The output images of the resized crop module have been obtained in a similar manner as the rotated images. There are three possible outputs that vary from the input image referred to by the number of pixels they were cropped to before resizing. These include 16, 20, and 24 pixels. The fourth possibility of no cropping has been omitted as this is simply a copy of the input image. The three tested possibilities are shown in Figure 7.5.

The original model uses resized cropping with bilinear interpolation which provides output images as shown in Figure 7.6 with a corresponding difference shown in 7.8d. An interpolation method which better resembles the methods used during hardware design and implementation is the nearest exact interpolation method Torchvision provides. The corresponding output images and differences to the hardware version can be found in respectively Figure 7.7 and Figure 7.8e. The images showing the differences compare the images that are the result of a 20 pixel crop.



(a) 24 pixel crop **(b)** 20 pixel crop **(c)** 16 pixel crop

Figure 7.5: Image being cropped to three possible sizes and being resized using the hardware module.

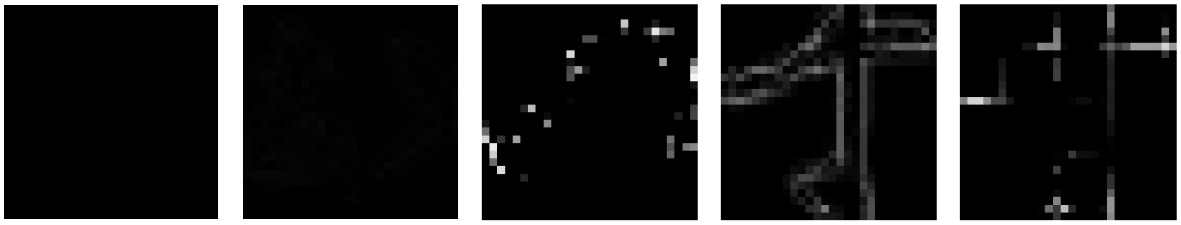


(a) 24 pixel crop **(b)** 20 pixel crop **(c)** 16 pixel crop

Figure 7.6: Image being cropped to three possible sizes and being resized using the Torchvision function with bilinear interpolation.

(a) 24 pixel crop **(b)** 20 pixel crop **(c)** 16 pixel crop

Figure 7.7: Example image being cropped to three possible sizes and being resized using the Torchvision function with nearest exact interpolation.

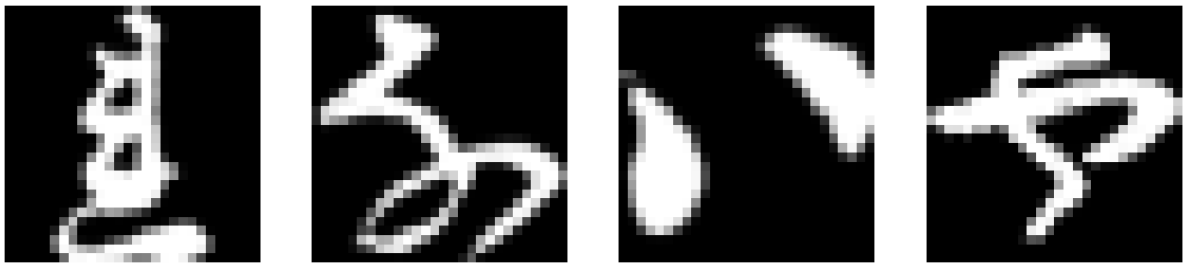


(a) Convolution **(b)** Gaussian Blur **(c)** Rotation **(d)** Resized crop bilinear **(e)** Resized crop nearest

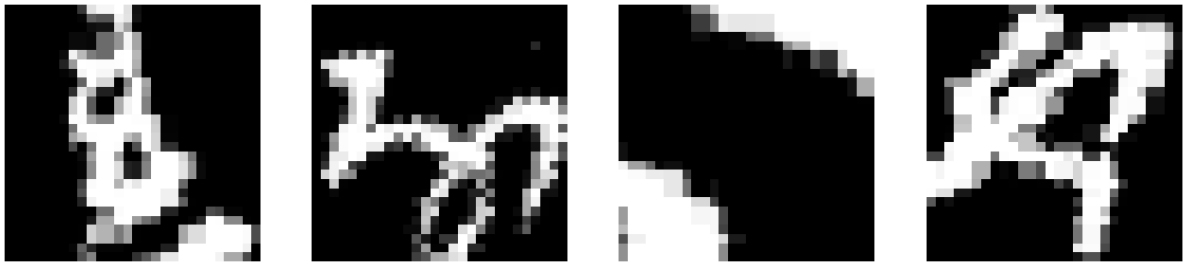
Figure 7.8: Absolute difference in pixel value between the hardware output and the Python result of the discussed augmentations.

7.2. Integrated System

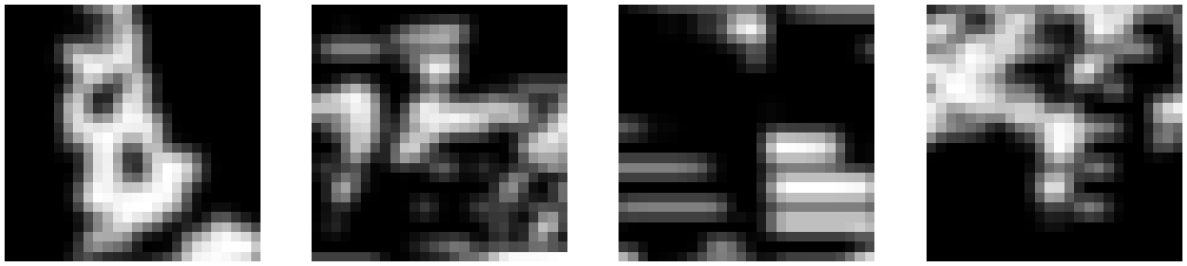
Several hardware modules have been combined into a system which pipelines several functionalities and is similar to the system shown in Figure 6.1, only missing the Gaussian blur module. The entire system is tested in the same way as the individual modules: images and a Gaussian kernel are written to shared memory, the image is processed and then written back to yet another memory system. The intermediate augmented images are also written in memory to read out via the Python interface. Figure 7.9 shows a number of images as they enter the programmable logic, after augmentation and after being completely processed. This system implementation which has enough BRAMs equiped to be able to process 40 images requires 25 BRAMs, 12.601 LUTs and 11.864 flipflops, which respectively correspond to 17.4% , 10.8% and 5.1% of the available resources.



(a) Images before processing



(b) Images after augmentation



(c) Images after convolution

Figure 7.9: Four different images from the KMNIST dataset before processing, after augmentation and after convolution.

7.3. Execution Time

The time it takes to execute the relevant Torch and Tochvision funtions has been measured by applying the same profiler as used in Section 4 to a file which performs these functions a number of times after which the average execution time was taken. Additionally, the time it takes to communicate with the programmable logic using the PYNQ library has been measured in a similar manner. This includes reading and writing control signals, but also writing kernels and images, and reading images. The measured results are presented in Table 7.1.

The time it takes for the hardware modules to perform their dedicated tasks has been estimated by counting the number of clock cycles the simulations require between the start signal (or the first valid pixel in some cases) and the last pixel being sent via the output. The execution time of these modules has been calculated by assuming a clock frequency of 100 MHz. These values represent the ideal case when one image is processed. Interruptions and additional smaller modules between these main modules can extend the time it takes for an image to be processed. Additionally, when a larger number of images is processed, the different modules can be pipelined decreasing the total time and the start-up period for some modules only occurs once.

Table 7.1: Timing comparison between the hardware modules obtained from simulations and Python functions and communication methods obtained from profiling.

	Functionality	Clock cycles	Simulation time (μs)	Profiling Execution time (μs)
Python	Write kernel			63
	Write image			2567.6
	Read image			2430.4
	Write control			14.95
	Read control			15.5
	Gaussian blur			1038.1
	Rotation			1316.8
	Resized Crop (bilinear)			608.833
	Convolution			132.8
Hardware	Read kernel	7	0.07	
	Read image	786	7.86	
	Write image	785	7.85	
	Gaussian blur	838	8.38	
	Rotation	787	7.87	
	Resized crop	787	7.87	
	Convolution	838	8.38	

Given these results, we can calculate the projected total processing time in the hardware and obtain an ideal speed-up factor. As the modules include extensive pipelining, after some set-up time, the modules can process an entire batch loaded to the BRAMs in a streamlined fashion. Importantly, the augmentations can occur simultaneously, and as the augmentation-unit sequentially generates output pixels of the augmented image, the CLA can be engaged simultaneously to load the buffers and perform the convolutions. Hence, the streamlined processing time should total the longest individual delay (ignoring the set-up time), which equals $8.38\mu s$ – i.e., the convolutional forward pass. Since the microcontroller equivalently requires $1038.1 + 1316.8 + 608.833 + 132.8 = 3096.533\mu s$, we posit that the hardware accelerated modules can achieve a speed-up of up to $370\times$.

This speed-up can be increased even further by raising the number of MACC-units that a single CLA oversees to perform the convolutions in parallel. Note that the memory block read-out is constrained by an output bandwidth of 32 bits per clock-cycle, which corresponds to *4 activations per clock cycle* when using the 8-bit unsigned integer format. Realistically, the present implementation of the CLA with an input bandwidth of 1 pixel per clock cycle would function sufficiently well, especially for intermediate layers where multiple channels need to be processed. In this latter case, four concurrent CLAs could be employed, each with an input bandwidth of 1 pixel per clock cycle. For layers beyond the input, the augmentation-unit is no longer useful, and the attainable speed-up is reduced to the convolution-time: this corresponds to a speed-up of $\frac{132.8}{8.38} = 15.85\times$ per CLA. Since the number of computations scales linearly with respect to the number of channels, so would the execution time on the microcontroller – whereas the CLA can employ at least four CLAs in parallel, potentially more if different input channels were to be stored across different memory blocks to increase throughput. Finally, the assumption of a clock speed of 100 MHz is fairly conservative, as it simply corresponds to the baseline clock frequency of the KRIA board. Of course, the extraneous communication overhead between the microcontroller and the programmable logic across the GPIO interface does not allow for the full exploitation of this speed-up anyhow, which is further discussed in the next chapter.

7.4. Self-Supervised Federated Learning

The self-supervised learning and federated learning algorithms provided by the Signal Processing and Algorithms subgroup were tested on a KRIA board without hardware acceleration to evaluate if the algorithms would be able to run on an client node as separate entity. It was possible to receive model parameters from the server, perform the self-supervised learning training algorithms and send back the updated parameters, which means FPGAs are certainly amenable to an end-to-end framework.

8

Discussion

8.1. Interpretation of Results

The accuracy of the Convolutional Layer Accelerator can be analysed by comparing the output images obtained from the on-chip implementation and the Python functions. These images seem visually identical which is confirmed by Figures 7.8a and 7.8b as the absolute differences are indeed very small. However, it should be noted that the Gaussian blur application of the CLA still has a smaller output dimension than is normal due to the absence of zero-padding the borders prior to the convolution with the Gaussian kernel.

The rotation module is able to rotate the input image with the four previously declared number of degrees, while maintaining the semantic information to the human eye. There are some slight changes between the Python and hardware generated images which mostly occur around the edges of the main shape shown. Torchvision also uses the nearest neighbour algorithm for its interpolation but the fine details might be different such as the rounding method or declared image center.

The resized crop module is also able to successfully crop and resize the input image to the three planned variations while still being recognisable. The most noticeable pixel differences between hardware and Python with bilinear interpolation seem to reside in smaller variations around the general shape as a result of the difference between the blurred edges and sharp edges resulting from the interpolation methods. When comparing the hardware output to the nearest neighbour interpolated variant of the Torchvision library, we observe only slight differences along some edges, possibly resulting from different rounding methods.

The integrated system excluding the Gaussian blur augmentation has been deployed successfully. Figure 7.9 shows that the augmented images appear more pixelated when compared to their equivalent input images which can be explained by the use of the nearest neighbour algorithms. Nevertheless, it is still possible to visually identify which input image corresponds to which augmented image. The CLA functions as expected on the first image but fails to correctly convolve the images that follow due to some remaining timing issues.

The main purpose of moving these functionalities from Python to hardware was to decrease the execution time. When comparing the execution time of the different augmentations and convolutions in Python to their time in hardware in Table 7.1, it is noticeable that the hardware modules are many times faster than their respective functions within Python. While designing these modules, the latency was kept as small as possible, and the designs are easily extendable. By using LUTs that store the indices for the rotation and resized crop, we are limited to a single output per clock cycle, but by duplicating the counters and LUTs we could increase throughput. However, the on-chip bandwidth is certainly not the problem, as the projected speed-up is certainly satisfactory; the primary constraint seems to be the extreme latency associated with the communication over the GPIO and AXI interface between the microcontroller and the programmable logic. As it stands, taking communication time into account, the accelerated pipeline would take about $1.75\times$ *longer* to complete its computations. However, less

than 5% of the execution time associated with this hardware implementation is actually utilised by the programmable logic, while the rest is used for communication.

8.2. Requirement Completion Analysis

Two lists of requirements were made in Section 2, one concerning the subsystem discussed in this thesis and the other concerning the entire proof-of-concept system. The requirements list concerning the Hardware and System Development will be reviewed here.

The system has shown to be capable of successfully running the self-supervised learning and federated learning algorithms, fulfilling requirement 1. Considering the system has to be able to share data to an external data and this data only consists of the model parameters to successfully perform the federated learning algorithm, requirements 2 and 3 have also been satisfied.

After identifying the computational bottlenecks of the employed algorithms, several modules were designed and some implemented on the KRIA board. Both the rotation and the resized crop augmentation modules work on the KRIA and can successfully substitute, to an extent, their respective Python functions with a smaller latency. The CLA module still has some timing issues but should only need little adaptation to function completely on the chip. Thus, requirement 4 is satisfied.

The hardware implemented modules have been tested on the KRIA and for this purpose data had to be transferred between the microprocessor and programmable logic. The hardware modules also had to be started via the Python interface. Both cases functioned well as can be seen when comparing the resulting images of the tests with their corresponding output images when using Python. Thus, the communication between the two sections proceeded fine and requirements 5 and 6 are met.

The system which currently includes all hardware implementable modules consists of 25 BRAMs, 12.601 LUTs and 11.864 flipflops. This is less than 20% of the available resources for all three instances, indicating that requirements 7, 8 and 9 are fulfilled. The number of used resources will increase when increasing the number of modules implemented on the programmable logic. Despite this, the amount of necessary resources can be somewhat controlled by varying the number of CLA modules implemented and the number of input images per round. The number of BRAMs required per arrangement can be estimated by using the formula created in Section 6.4. The number of LUTs and flipflops is more difficult to estimate precisely but will also scale as more modules are implemented within the programmable logic.

There were also two trade-off requirements made for the subsystem. These requirements have been taken into account during the entire design and implementation process. The second of these requirements could be considered as completed, as the actual implementation can be easily modified by increasing and decreasing the number of designed modules.

8.3. Future Work

The first steps towards a self-supervised learning hardware accelerator have been made but its implementation has not been completed entirely. Whereas the rotation and resized crop modules have been successfully implemented on the KRIA, the overarching connectivity between the modules should still be designed to ensure proper, controllable, and streamlined functioning. As a result, the hardware acceleration modules have not yet been tested in combination with the self-supervised learning algorithms, though the Signal Processing & Algorithms group has performed prospective experiments with quantized values for the weights and activations similising our formats. A direct continuation of the project would consist in combining the remaining modules with the current implemented design to obtain a system as discussed in Section 6.3. The effects of this hardware implementation on the latency and accuracy can then be duly evaluated. This process can be followed by an analysis and a more detailed design space exploration to see what solutions can be offered to obtain the desired effects.

One of the aspects that could be further analysed is the effect of moving the augmentations to the hardware. A possible decrease of accuracy could be the result of the simplified methods of generating the augmented images in hardware, using pre-defined LUTs to match the original pixels to their locations within the augmented image saves a lot of time when compared to performing complex computations but it might also decrease the quality of the images. Another simplification was the transition from a

randomly located resized crop to a central resized crop. Since non-contrastive methods essentially attempt to derive the expected value of some semantic embedding across augmentations, limiting and simplifying augmentations might influence accuracy. Additionally, a major part of the BYOL and SimSiam models is introducing small random changes. Limiting the large range of possibilities to only four fixed values for rotation and resized crop and to one type of Gaussian blur greatly decreases the number of variations that an image can obtain. It would be useful to see if this has any major effects, otherwise it might be useful to increase this number of fixed values or replace the LUT-centered design to a real-time arithmetic unit. During this project, only three augmentations were considered that corresponded to the augmentations used by the other subgroup. It could be interesting to look more into different augmentation methods that have a beneficial effect on the accuracy while being simple to implement as hardware module, due to the extensibility of the CLA.

One major issue the current design contains is the enormous communication overhead. This problem is mainly caused by the amount of time it takes to transfer data between the microprocessor and the programmable logic. Additionally, no other actions can be performed while data is being transferred. On the hardware side, the modules process images too fast to be used at the same time as the microprocessor without causing problems. Meanwhile, the actions of the microprocessor are halted while data is transferred, effectively causing quite a large disruption during execution of the program and obstructing parallel execution. The briefly mentioned data transfer method called Direct Memory Transfer or Direct Memory Access (DMA) could prove to be a promising alternative. This method, namely, has a much larger throughput than the currently used memory mapped system: up to roughly 300 MB/s and 400 MB/s for transferring from memory-mapped to stream and vice versa [24]. It is also possible to perform these transfers while the microcontroller continues with other tasks, enabling parallelisation. It would be recommended to explore the possibilities of DMA more in order to decrease the large communication overhead.

Another method to decrease the considerable amount of communication overhead would be to decrease the number of times data has to be transferred between the microprocessor and the programmable logic. This can be done by moving more sections of the model to the programmable logic. At the moment, the output of the augmentations and intermediate activations have to be transferred back to the microprocessor in order to be able to perform inter-layer batch normalisation and backpropagation. The implementation of backpropagation was kept out of the scope of the project due to time constraints but might be worth investigating as it greatly decreases the necessary communication between the two subsystems. However, adding the backpropagation to the programmable logic would significantly increase the area use, possibly delimiting the achievable speed-up of the CLA. This might be no problem for very small CNNs but will likely cause problems for CNNs of more standard size. Nevertheless, the adoption of on-chip training might be worthwhile.

Finally, to provide a more general perspective on the prospect of ‘self-supervised federated learning at the edge’: if the mentioned optimisations were to be pursued, FPGAs could prove to be perfect for the purposes of accelerating the immense computational requirements underpinning the self-supervised learning algorithms. A major bottleneck in terms of latency has proven to be backpropagation but also, very likely, the inter-layer batch normalisation. Both require the storage of intermediate activations, but, importantly, batch normalisation obstructs the network from feeding its activation further forward until all intermediate outputs have been obtained for a given (mini-)batch. This seriously impedes continuous, pipelined streaming that the CLA would be able to achieve otherwise. Hence, investigating layer normalisation techniques or even easier and more immediate normalisation methods, such as the L1 Filter-Response Normalisation (or L1-FRN) as proposed in [25], might prove salient – which would additionally eliminate communication overhead as the programmable logic would no longer need to rely on the microcontroller to perform the computation itself.

9

Conclusion

The advancement of self-supervised learning applications to edge devices within a federated learning set-up is thwarted by the extensive computational overhead associated with training these models. This project aimed to converge the algorithms discussed in [2] with the possibility of hardware acceleration. To this end, two subgroups were created, one focusing on the implementation of these algorithms within Python and the other focusing on the hardware acceleration. This thesis described the efforts of the second subgroup, discussing the design and implementation of several hardware modules.

The BYOL self-supervised learning algorithm as implemented by the Signal Processing and Algorithms subgroup was analysed in order to find the most time-expensive sections of the model. The results of this analysis included the observation that the augmentations, the encoder consisting of convolutional layers and the backpropagation required the most execution time. The augmentations and convolutional layers were chosen to be transferred to hardware due to their implementations being the most realistic given the time constraints of the project.

A Convolutional Layer Accelerator has been designed by combining the functionality of pixel buffers and a Multiply-and-Accumulate unit controlled by an overarching FSM. This module can be slightly modified to apply a Gaussian blur (or, theoretically, any spatial convolution-based augmentation) to the input images as part of the augmentations performed on these images. Additionally, a rotation and a resized crop module have been designed using a method which sequentially outputs the pixels of the augmented image by reading a pixel from the original image within a memory buffer mapped to the augmented image. This allows for a quick and simple implementation using LUTs to allow for a small range of random augmentations. The randomisation of these augmentations has been implemented by using a number of LFSRs as pseudo-random number generators.

The augmentation modules and the CLA have been implemented and the outputs of these modules are similar to the outputs of their corresponding Python functions with the exception of some small deviations most likely caused by simplifications made within the hardware modules. The time it takes to perform the functions on the hardware modules, obtained using simulations, is many times smaller than the time it takes to perform them within Python. However, the communication overhead as a result of the large amounts of data required to be transferred between microprocessor and programmable logic seems to become a large hurdle, increasing the expected time for the algorithms to be executed.

Despite this setback, the predefined requirements have all been satisfied. Direct continuation of the project would consist of implementing all designed modules within one system which can be used in combination with the self-supervised learning algorithms. After this, the effect of the augmentations can be analysed and solutions can be examined to decrease the communication overhead by finding other methods of data transfer such as DMA and moving more modules to the programmable logic at the cost of chip area.

Bibliography

[1] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, and M. Valko, "Bootstrap your own latent: A new approach to self-supervised learning," 2020.

[2] J. Shi, Y. Wu, D. Zeng, J. Tao, J. Hu, and Y. Shi, "Self-supervised on-device federated learning from unlabeled streams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, p. 4871–4882, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2023.3274956

[3] F. Murtagh, "Multilayer perceptrons for classification and regression," *Neurocomputing*, vol. 2, no. 5, pp. 183–197, 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0925231291900235

[4] R. M. Schmidt, "Recurrent neural networks (rnns): A gentle introduction and overview," *arXiv preprint arXiv:1912.05911*, 2019.

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

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 2008. [Online]. Available: https://books.google.nl/books?id=qMLUIsgCwvUC

[8] F. L. Bauer, "Computational graphs and rounding error," *SIAM Journal on Numerical Analysis*, vol. 11, no. 1, pp. 87–96, 1974. [Online]. Available: http://www.jstor.org/stable/2156433

[9] X. Chen and K. He, "Exploring simple siamese representation learning," 2020.

[10] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging," *CoRR*, vol. abs/1602.05629, 2016. [Online]. Available: http://arxiv.org/abs/1602.05629

[11] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[12] T. Clanuwat, M. Bober-Irizar, A. Kitamoto, A. Lamb, K. Yamamoto, and D. Ha. (2018) Deep learning for classical japanese literature.

[13] R. Kern. Line profiler. [Online]. Available: https://pypi.org/project/line-profiler/

[14] A. Al-Dujaili and S. A. Fahmy, "High throughput image filters on fpgas," *ArXiv e-prints*, vol. arXiv:1710.05154, 2017. [Online]. Available: http://arxiv.org/abs/1710.05154

[15] H. Kim, J. Park, C. Lee, and J.-J. Kim, "Improving accuracy of binary neural networks using unbalanced activation distribution," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 7862–7871.

[16] T. maintainers and contributors, "Torchvision: Pytorch's computer vision library," https://github.com/pytorch/vision, 2016.

[17] AMD: Technical Information Portal. Efficient shift registers, lfsr counters, and long pseudo-random sequence generators (xapp052). [Online]. Available: https://docs.amd.com/v/u/en-US/xapp052

[18] ——, “Kria kv260 vision ai starter kit user guide (ug1089).” [Online]. Available: https://docs.amd.com/r/en-US/ug1089-kv260-starter-kit/Accelerated-Application-Package-Selection

[19] ——. Zynq® ultrascale+™ mpsoc data sheet: Overview (ds891). [Online]. Available: https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview

[20] Advanced Micro Devices, “Vivado Design Suite (2023.2),” 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[21] ——, “PYNQ: Python productivity for Adapting Computing platforms,” 2024. [Online]. Available: http://www.pynq.io/

[22] AMD: Technical Information Portal. Axi block ram (bram) controller v4.1 product guide(pg078). [Online]. Available: https://docs.amd.com/v/u/en-US/pg078-axi-bram-ctrl

[23] ——. Block memory generator v8.4 product guide (pg058). [Online]. Available: https://docs.amd.com/v/u/en-US/pg058-blk-mem-gen

[24] ——. Axi dma v7.1 logicore ip product guide. [Online]. Available: https://docs.amd.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide

[25] J. Lu, C. Ni, and Z. Wang, “Eta: An efficient training accelerator for dnns based on hardware-algorithm co-optimization,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 10, pp. 7660–7674, 2023.