

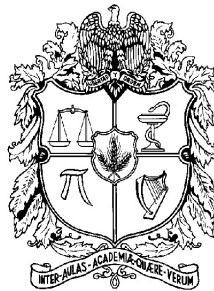
**A New Framework for Training a CNN with a Hardware-Software
Architecture**

A thesis submitted in partial fulfillment of the requirements for the degree of:
PhD. in Electrical Engineering

Presented to:
Coordinación de Posgrados de la Vicedecanatura Académica

By:
Dorfell Leonardo Parra Prada

Advisor:
PhD. Carlos Ivan Camargo Bareño



UNIVERSIDAD NACIONAL DE COLOMBIA
Sede Bogotá

Facultad de Ingenierías
Agosto, 2023

Prólogo

“El romper de una ola no puede explicar todo el mar”

Vladimir Nabokov.

I. Propósito

En mi opinión la construcción personal debe ser tan importante como la académica. Nos alarman los indicadores de deserción, las tasas de suicidio, la droga adicción, etc., pero somos incapaces de mitigar sus causas. Además, mantenemos la idea de que el bienestar sólo depende de la acumulación de bienes materiales, e ignoramos el alimento del ser, del alma...

En mi desarrollo personal encontré que mi propósito actual es aprender a ser feliz. Y que mi felicidad no debe depender de lo material:

*“Não se pode comprar o vento, não se pode comprar o sol,
não se pode comprar a chuva, não se pode comprar o calor,
não se pode comprar as nuvens, não se pode comprar as cores,
não se pode comprar minha alegria, não se pode comprar minhas dores”*

Latinomérica, Calle 13.

Además, no importa si no encajamos en un molde, siempre y cuando respetemos y aceptemos a los demás. Ahora bien, estimado lector, como lo dice Na Morales en su canción “Fui y Volví”, ... *yo he visto el mundo con mis ojos miopes...*, por lo que mi mirada es limitada y no pretendo que mis palabras sean verdades absolutas.

II. Agradecimientos

Agradezco a Dios por mi familia y por esta oportunidad de crecimiento personal y profesional.

También agradezco a mi familia, que siempre ha estado ahí para mí, ellos me motivan y apoyan a pesar de mis locas y necias ideas.

A mi director, el profe Carlos, por su apoyo incondicional y su don de gente, muchas gracias.

A las personas que vinieron, a las que se fueron, a las que se quedaron, aprendí mucho de ustedes, reí, lloré, viví.

Cada uno de nuestros recuerdos forma parte de la colcha de retazos que narra mi vida.

III. Ahora sí el *prólogo*.

Creo que el desarrollo del país no debe depender del extractivismo de recursos finitos. Necesitamos una economía multisectorial orientada a la agricultura e innovación tecnológica para la solución de problemas en las regiones. Vivimos en una dinámica que genera toneladas de basura a diario y tenemos la oportunidad de ser un ejemplo de conservación de la biodiversidad, y respeto por los páramos, el agua y la vida.

Por esta razón, uno de los propósitos de este trabajo es impulsar el desarrollo de tecnologías que contribuyan al crecimiento del país. Específicamente en este trabajo encontrarás la apropiación tecnológica de redes neuronales convolucionales (CNNs) y Sistemas on Chip (SoC). Además se documenta la manera en que se resolvieron los problemas y se cumplieron los objetivos propuestos. Anheló, que lo aquí presentado no termine en los anaqueles olvidados de nuestra biblioteca, si no que pueda ser de utilidad para cualquier mente ansiosa que quiera apropiarse este modesto trabajo.

Abstract

A New Framework for Training a CNN with a Hardware-Software Architecture

Facial Expression Recognition (FER) systems classify emotions by using geometrical approaches or Machine Learning (ML) algorithms such as Convolutional Neural Networks (CNNs). However, designing these systems could be a challenging task that depends on the data set's quality or the designer's expertise. Moreover, CNNs inference requires a large amount of memory and computational resources, making it unfeasible for low-cost embedded systems. Hence, although GPUs are expensive and have high power consumption, they are frequently employed because they considerably reduce the inference time compared to CPUs.

On the other hand, SoCs implemented in FPGAs could employ less power and support pipelining. However, the floating point representation may result in intricate and larger designs that are only suitable for high-end FPGAs. Therefore, custom hardware-software architectures that maintain acceptable performance while using simpler data representations are advantageous.

To address these challenges, this work proposes a design methodology for CNN-based FER systems. The methodology includes the preprocessing, the Local binary pattern (LBP), and the data augmentation. Besides, several CNN models were trained with TensorFlow and the JAFFE data set to validate the methodology. In each test, the relationship between parameters, layers, and performance was studied, as were the overfitting and underfitting scenarios. Furthermore, this work introduces the model M6, a single channel CNN that reaches an accuracy of 94% in less than 30 epochs. M6 has 306.182 parameters in 1.17 MB.

In addition, the work also employs the quantization methodology from TensorFlow Lite (tflite), to compute the inference of a CNN using integer numbers. M6's accuracy dropped from 94.44% to 83.33% after quantization, the number of parameters increased from 306.182 to 306.652, and the model size decreased almost 4× from 1.17 MB to 0.3 MB.

Also, the work presents a custom hardware-software architecture to accelerate CNNs known as the FER SoC, which reproduces the main tflite operations in hardware. Hence, as the integer numbers are fully mapped to hardware registers, the accelerator results will be similar to their software counterparts. The architecture has been tested on a Zybo-Z7 development board with 1 GB RAM and the Zynq7 device XCZ7020-CLG400. Moreover, it was observed that the architecture got the same accuracy but was 20% slower than a laptop equipped with an AMD CPU with 16 threads, 16 GB of

RAM and a Nvidia GTX1660Ti GPU. Therefore, it is recommended to assess whether the trade-off between quantization and inference time is worth it for the target application.

Lastly, another contribution is the framework for CNNs' training in custom hardware-software architectures known as Resiliency. It has been used to train and run the inference of the single-channel M6 model. Resiliency provides the design files needed as well as the Pynq 2.7 image created for running ML frameworks such as TensorFlow and PyTorch. Although the training time was slow, the accuracy and loss were consistent to traditional approaches. However, the execution time could be improved by utilizing bigger FPGAs with MPSoCs like the Zynq Ultrascale family.

Keywords: FER, CNN, FPGA, HNN.

Resumen

Nuevo *framework* para entrenar CNNs con una arquitectura hardware-software

Los sistemas de reconocimiento de expresiones faciales (FER) clasifican emociones usando estrategias geométricas o algoritmos de Machine Learning (ML) como redes neuronales convolucionales (CNNs). Sin embargo, el diseño de estos sistemas es una tarea compleja que depende de la calidad del set de datos y la experiencia del diseñador. Además, la inferencia de las CNNs requiere recursos de memoria y cómputo que hacen inviable el uso de sistemas embebidos de bajo costo. Igualmente, aunque las GPUs son costosas y presentan un alto consumo de potencia, se utilizan frecuentemente porque reducen el tiempo de ejecución en comparación a las CPU.

Por otro lado, los sistemas on-chip (SoCs) implementados en FPGAs emplean menos potencia y soportan cómputo en paralelo. No obstante, representaciones numéricas como punto flotante pueden resultar en diseños complejos sólo adecuadas para FPGAs de gama alta. Por esta razón, el uso de arquitecturas de hardware-software que emplean representaciones numéricas sencillas y mantienen un desempeño aceptable son favorables.

Para afrontar estos desafíos, este trabajo propone una metodología de diseño para sistemas FER basados en CNNs. La metodología incluye el preprocesamiento, el patrón local binario (LBP), y la aumentación de datos. Asimismo, para validar la metodología varios modelos CNNs fueron entrenados con TensorFlow y el set de datos JAFFE. En cada test, se estudia la relación entre los parámetros, las capas y el desempeño, el subentrenamiento y el sobreentrenamiento. Además, este trabajo introduce un modelo CNN de un canal llamado M6 que alcanza una exactitud de 94% en menos de 30 épocas. M6 tiene 306.182 parámetros y emplea 1.17 MB.

El trabajo también utiliza la estrategia de cuantización de TensorFlow Lite (tflite) para computar la inferencia de la CNN empleando números enteros. Después de la cuantización, la exactitud de M6 se redujo de 94.44% a 83.33%, el número de parámetros aumentó de 306.182 a 306.652, y el tamaño del modelo se redujo aproximadamente 4 veces pasando de 1.17 MB a 0.3 MB.

Igualmente, el trabajo presenta el FER SoC, una arquitectura de hardware-software para la aceleración de CNNs que reproduce las operaciones principales de tflite en hardware. En este caso, como los registros en hardware soportan las operaciones enteras, los resultados del acelerador son similares

a la contraparte software. El FER SoC fue implementado en el sistema de desarrollo Zybo-Z7, el cuál emplea 1 GB RAM, y la FPGA Zynq XCZ7020-CLG400. Además, se observó que la arquitectura obtuvo la misma exactitud que una laptop con una CPU AMD de 16 threads, 16 GB de RAM, y la GPU de Nvidia GTX1660Ti, pero fue 20% más lenta. Por lo que se recomienda evaluar sí el intercambio entre la quantización y el tiempo de ejecución es suficiente para la aplicación objetivo.

Por último, otra contribución del trabajo es el framework Resiliency, el cuál permite el entrenamiento y la inferencia de modelos CNN de un solo canal. Resiliency, provee los archivos de diseño necesarios y la imagen Pynq 2.7 creada para ejecutar los frameworks de ML TensorFlow y PyTorch. Aunque el tiempo de entrenamiento fue lento, la exactitud y la pérdida son consistentes con las estrategias tradicionales. Sin embargo, el tiempo de ejecución puede ser mejorado usando FPGAs de gama alta con MPSoCs como las Zynq Ultrascale.

Palabras claves: FER, CNN, FPGA, HNN.

Contents

	Pag.
Contents	viii
List of Figures	xi
List of Tables	xii
1 Facial Expression Recognition (FER) Systems	3
1 Facial Expression Recognition (FER) Systems	4
1.1 CNN-based FER Systems	4
2 Data sets for FER systems	7
3 Basic data set preprocessing for FER systems	8
4 Extended data set preprocessing for FER systems	10
4.1 Local Binary Pattern	10
4.2 Data Augmentation	11
5 Designing a CNN-based FER system	13
5.1 Testing model M1	15
5.2 Testing model M2	16
5.3 Testing model M3	18
5.4 Testing model M4	19
5.5 Testing model M6	20
6 Discussion	21
7 Conclusion	21
2 Hardware Neural Networks (HNN)	22
1 Quantization Aware Training	23
2 Hardware Platform	25

3	Hardware-software Development	29
3.1	Base SoC Design	29
3.2	TfLite Core	31
3.3	Conv Core	34
3.4	MaxPooling Core	36
3.5	Dense Core	37
3.6	Additional functions	39
4	Putting all together: the Hardware Neural Network Accelerator	41
4.1	FER SoC design	41
4.2	FER SoC testing	45
5	Discussion	47
6	Conclusion	47
3	Resiliency: A Framework for Training a CNN with a custom Hardware-Software (HW-SW) Architecture.	48
1	Custom hardware-software architectures aimed to ML algorithms	49
2	Creating a Pynq image for Zybo-Z7	51
2.1	Create the Board Directory	52
2.2	Create the Board Support Package (BSP) (30 min approx.)	53
2.3	Create the image (2h30 min approx.)	55
2.4	Copy image to micro SD card (20 min approx.)	58
3	Testing ML Frameworks on the hw-sw architectures proposed	59
3.1	Tensorflow 2.5 in Zybo-Z7 running Pynq 2.7	59
3.1.1	Installing Python3.7 (50 min approx.)	59
3.1.2	Create a virtual environment for Python3.7 (5 min approx.)	59
3.1.3	Install TensorFlow in the virtual environment (> 5h approx.)	60
3.1.4	Create the IPython Kernel (1h approx.)	61
3.1.5	Testing Tensorflow 2.5 running on Zybo-Z7 with Pynq 2.7	61
3.2	PyTorch 1.8 in Zybo-Z7 running Pynq 2.7	63
3.2.1	Installing Python3.7 (50 min approx.)	63
3.2.2	Create a virtual environment for Python3.7 (5 min approx.)	63
3.2.3	Install PyTorch in the virtual environment (> 2h approx.)	63
3.2.4	Create the IPython Kernel (1h approx.)	64
3.2.5	Testing PyTorch 1.8 running on Zybo-Z7 with Pynq 2.7	64
3.3	Testing model M6 in Zybo-Z7 running Pynq 2.7	66
4	Resiliency: A Framework for Training a CNN with a custom Hardware-Software Architecture.	68

5	Discussion	71
6	Conclusion	72
	Bibliography	75
	A State of the art of the Hardware Neural Networks (HNNs)	81
1	Systematic Literature Review Method	81
1.1	Research Questions (RQ)	82
1.2	Search Process	82
1.3	Study selection	83
1.4	Quality assessment (QA)	84
1.5	Data collection	84
1.6	Data analysis	85
2	Results	85
2.1	Search Results	85
2.2	Quality evaluation of the HNN works.	85
2.3	Quality factors	85
3	Discussion	86
4	Conclusions	88
5	Research Questions	88
6	Hypothesis	89
	B Useful Scripts	90
	C Complementary Resources	98

List of Figures

1.1	Examples of face action units (AUs). Taken from [1].	5
1.2	Examples of CNN-based FER systems architectures proposed in literature.	6
1.3	Examples of data sets used in FER systems.	7
1.4	Acquisition setup used for JAFFE. Taken from [2].	8
1.5	JAFFE Pre-processing for FER. Adapted from [3].	9
1.6	Facial Point Annotations. Taken from [4].	9
1.7	LBP encoding example. Adapted from from [5], [6].	10
1.8	LBP methods comparison with JAFFE.	11
1.9	Examples of data augmentation with JAFFE.	12
1.10	Tests with the M1 model.	16
1.11	Tests with the M2 model.	17
1.12	Tests with the M3 model.	18
1.13	Tests with the M4 model.	19
1.14	Tests with the M6 model.	20
2.1	TfLite Quantization Aware Training Example.	23
2.2	Zybo-Z7 Development board. Taken from [7].	25
2.3	Zynq APSoC Architecture Simplified. Adapted from [8].	26
2.4	PL architecture. Taken from [9].	27
2.5	BRAM structure. Taken from [10].	28
2.6	Basic DSP48 Slice. Taken from [11].	28
2.7	Basic ARM SoC.	29
2.8	Vivado Block Design for the Basic ARM SoC.	29
2.9	TfLite Core.	32
2.10	TfLite Core Simulation.	33
2.11	Conv Core.	35

2.12	Conv Core Simulation.	35
2.13	MaxPooling Core.	37
2.14	Dense Core.	39
2.15	FER SoC.	41
2.16	Vivado Block Design for the FER SoC.	42
2.17	Memory map for the FER SoC.	43
2.18	FER SoC place and route.	44
2.19	FER SoC power consumption.	44
2.20	Running the inference of a CNN-based FER system in the accelerator.	46
3.1	Examples of AI desktop pets: Vector by DDL and EMO by Living AI.	49
3.2	Hardware devices for acceleration.	50
3.3	Vitis AI and hardware platforms by Xilinx.	50
3.4	Pynq development environment. Taken from [12].	51
3.5	Board Support Package adapted for the Zybo-Z7 board.	53
3.6	Vivado Block design for the BSP targetted to the Zybo-Z7 board.	54
3.7	Block diagram of base overlay defined in <i>base.tcl</i>	56
3.8	Vivado Block design of base overlay defined in <i>base.tcl</i>	57
3.9	Pynq 2.7 running on Zybo-Z7.	58
3.10	Tensorflow running on Zybo-Z7 with Pynq.	61
3.11	TensorFlow <i>model.fit</i> error.	62
3.12	PyTorch 1.8 running on Zybo-Z7 with Pynq.	65
3.13	Hw-sw architecture for training the model M6 with PyTorch 1.8 in Pynq 2.7.	67
3.14	Resiliency framework overview.	70
B.1	Notebook for M6 model in PyTorch installed on Pynq 1.8 running on the Zybo-Z7 board.	95
B.2	Notebook for M6 model in PyTorch installed on Pynq 1.8 running on the Zybo-Z7 board.	96
B.3	Notebook for M6 model in PyTorch installed on Pynq 1.8 running on the Zybo-Z7 board.	97
C.1	TfLite Quantization Aware Training. Taken from [13].	98
C.2	Zynq APSoC Architecture. Taken from [8].	99
C.3	PL architecture. Taken from [9].	99
C.4	Validation of results. Comparison between the Numpy and the SoC.	100
C.5	M6 model trained with PyTorch 1.8 in Pynq 2.7.	100

List of Tables

1.1	Architectures of some CNN-based FER systems and their accuracy for JAFFE.	13
2.1	Quantized models with comparable accuracy to floating point.	24
2.2	Specification for the quantized layers used in M6.	24
2.3	Metrics for model M6 in float point and after quantization.	24
2.4	XC7Z020-1CLG400C Specification.	28
2.5	FER SoC Utilization.	43
2.6	M6 inference performance comparison.	47
3.1	Loss values for laptop and the hw-sw architecture proposed.	65
3.2	M6 training performance comparison.	67
A.1	Selected journals, symposiums and conference proceedings.	83
A.2	Systematic Review of HNN Studies.	86
A.3	Quality evaluation of the HNN studies.	87
A.4	Average Quality Scores (QS) for studies by publication date.	87

INTRODUCTION

Facial expression recognition (FER) systems are essential for applications such as human-computer communication, animation, psychiatry, and automobile safety [14], [5], [6]. However, due to the diversity in face phenotypes and variability in the structure of facial expressions from person to person, designing FER systems remains challenging. Additionally, the image acquisition process is subject to factors that increase the complexity of the classification problem, including poor illumination, face rotations, and noise (e.g., hair, glasses). Furthermore, there is a lack of guidelines and code examples in the literature to provide training to designers. FER systems typically use complex multi-channel Machine Learning (ML) architectures suitable for implementation with high-performance hardware, but not with embedded devices. These ML algorithms include Convolutional Neural Networks (CNN), known for their potential to extract features from the input data. CNNs are used in several applications, like object detection and semantic segmentation [15], [16]. Also, they have been shown to increase the performance of the traditional FER applications [17], [18].

One of the open-source frameworks for developing ML applications is TensorFlow [19]. When it's used for training CNN models, the parameters obtained are represented by floating-point numbers. Unfortunately, keeping double precision on devices with limited resources could be unfeasible. One of the strategies to successfully execute the inference of CNNs on those devices is to represent the model parameters and computations with integer numbers. This could be achieved by means of techniques such as quantization, available in the TensorFlow's library, tfLite [20]. Nonetheless, the process implies a trade-off between the model's accuracy and its behavior in hardware.

On the other hand, embedded systems are commonly bound to the inference stage of a ML application, because the training phase is complex and entails a large number of memory and computational resources. For instance, [21] presents the challenges that face IoT devices when used in ML scenarios. However, training ML algorithms locally could be relevant in scenarios where new data samples must be considered at run-time or where previously trained models will be retrained, as in the Transfer Learning technique [22]. But the reports on training are scarce and evidence a gap in frameworks for training CNNs in custom hardware-software architectures.

To work around the drawbacks presented above, the following objectives have been proposed:

General Objective

Propose a framework that allows to train and execute a CNN in an embedded system (hardware-software), being able to use custom or the most popular software libraries. The framework will create the files needed to describe the architecture and will provide the tools for simulating and implementing it. Moreover, the framework will give advice about the hardware-software trade-offs needed to improve

the performance.

Specific Objectives

- Design and implement a basic FER application based on CNNs by means of software libraries.
- Design and evaluate accelerators for implementing the CNN feed forward execution in hardware.
- Design and validate a hardware-software architecture that implement the training algorithm and fits in an autonomous embedded system.
- Design and implement a framework that integrates the previously defined objectives and employs a design space exploration to propose the most cost-effective hardware.
- Evaluate the framework performance by means of metrics used in the literature such as the neural network accuracy and the execution time.

The thesis has the following structure: Chapter 1 presents a design methodology for a single-channel CNN-based FER system, along with the image pre-processing, the data augmentation, and the M6 CNN model trained with the JAFFE data set [23]. Then, chapter 2 describes the quantization process applied to the M6 model using tflite and the IP cores designed for the custom hardware-software architecture to accelerate its inference in hardware. Lastly, chapter 3 shows a base system designed for Pynq that runs a Jupyter server on a Zybo-z7 development board and is capable of running the ML frameworks TensorFlow and PyTorch. In addition, the Resiliency framework is introduced. Resiliency puts together all the approaches used in this work, provides the design resources needed to train CNNs on embedded systems, and gives guidelines for the design space exploration.

Facial Expression Recognition (FER) Systems

Facial expression recognition (FER) systems are examples of interaction technology based on Machine Learning (ML) and are fundamental for applications such as Human-Computer Communication, animation, psychiatry, automobile safety, etc. [14], [24], [5]. However, the design of FER systems is a challenging task due to the different phenotypes around the world.

Besides, the acquisition process is subject to illumination conditions, face rotation, noise (e.g., hair, glasses), etc., that increase the complexity of the classification problem. Hence, it is recommended to preprocess the images before using ML algorithms like Convolutional Neural Networks (CNNs). According to the literature, CNNs have increased the performance of FER systems, making them an interesting study topic that will be covered in this chapter.

First, the traditional FER system structure will be presented in section 1. Subsection 1.1 introduces the CNN-based FER systems; a recent strategy to integrate ML and traditional FER systems.

Section 2 shows the data sets used for training and validating the FER systems proposed in the literature. Section 3 depicts the basic data set preprocessing, while section 4 depicts the extended preprocessing: Local Binary Pattern (LBP) and Data Augmentation (DA).

The design process of a single channel CNN-based FER system is described in section 5, including some of the models tested and their learning curves. The section concludes with M6, a feasible model for running inference on embedded systems. Lastly, the discussion and conclusions can be found in sections 6 and 7.

1 Facial Expression Recognition (FER) Systems

Traditional FER systems had been implemented based on the detection of facial actions associated with prototypical expressions like happiness, sadness, anger, surprise, disgust, and fear. For instance, the Facial Action Coding System (FACS) proposed in 1977 by Ekman and Friesen [14], associated these facial actions with expressions. In FACS, individual or group changes in the facial muscles are coded as Action Units (AUs). For instance, the action of raising the inner brow, caused by the Frontalis and Pars Medialis muscles, is known as AU1 while AU19 is the action of “Tongue Out” [1]. Moreover, AUs can be additive when they appear independently and non-additive when different AUs modify each other. Some AUs for the upper and lower face are shown in Figure 1.1.

Unfortunately, FER systems based on traditional methods present drawbacks due to the different ways AU changes from individual to individual, even in the same expression. Moreover, the image acquisition process could lead to different conditions of illumination, exposure, face angle, rotation, etc. that could affect the classification accuracy of the FER system [17].

On the other hand, CNNs have become a feasible alternative to implementing FER systems, mainly because they are able to extract features from the input data and train their parameters by using those feature maps [25], [5], [6], [17].
















1.1 CNN-based FER Systems

According to the literature, a CNN-based FER system can be composed of input data preprocessing and the classification architecture. This architecture could include one or more channels with the goal of increasing accuracy [25], [5], [6], [17]. For example, one channel could be a CNN while the other used a ML algorithm such as a Support Vector Machine (SVM) or another CNN.


An extensive survey on FER can be found in [18]. It covers the databases available and their preprocessing, along with the deep networks used: CNNs, Deep Belief Networks (DBN), Deep Autoencoder (DAE), Recurrent Neural Network (RNN), and General Adversarial Network (GAN). Moreover, the work points out that deep networks can underperform if the training data is insufficient. Other works found are listed below.

In [25] authors proposed a single channel system called I^2CNN , which is shown in figure 1.2a. I^2CNN uses five convolutional layers and one SVM and reaches an accuracy of 75% with JAFFE and 98.3% with CK+, both data sets using six emotions. On the other hand, [5] presented a two-channel approach named WMDNN (see figure 1.2b) that integrates two CNNs: a partial VGG16 network and a shallow CNN. Its input were 72×72 pixels images from a Local Binary Pattern (LBP), and it achieved accuracy of 92.21% with JAFFE and 97.02% with CK+.

Another example of a two-channel FER system is [6] shown in figure 1.2c. This work integrates two CNNs: one using geometric features and the other using appearance features from LBP images. This system achieves 91.27% with JAFFE and 96.46% with CK+.

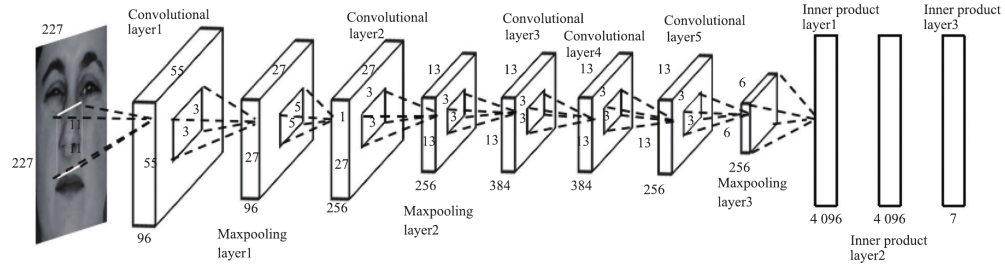
<i>NEUTRAL</i>	AU 1	AU 2	AU 4	AU 5
				
Eyes, brow, and cheek are relaxed.	Inner portion of the brows is raised.	Outer portion of the brows is raised.	Brows lowered and drawn together	Upper eyelids are raised.
AU 6	AU 7	AU 1+2	AU 1+4	AU 4+5
				
Cheeks are raised.	Lower eyelids are raised.	Inner and outer portions of the brows are raised.	Medial portion of the brows is raised and pulled together.	Brows lowered and drawn together and upper eyelids are raised.
AU 1+2+4	AU 1+2+5	AU 1+6	AU 6+7	AU 1+2+5+6+7
				
Brows are pulled together and upward.	Brows and upper eyelids are raised.	Inner portion of brows and cheeks are raised.	Lower eyelids cheeks are raised.	Brows, eyelids, and cheeks are raised.

(a) Upper face AUs.

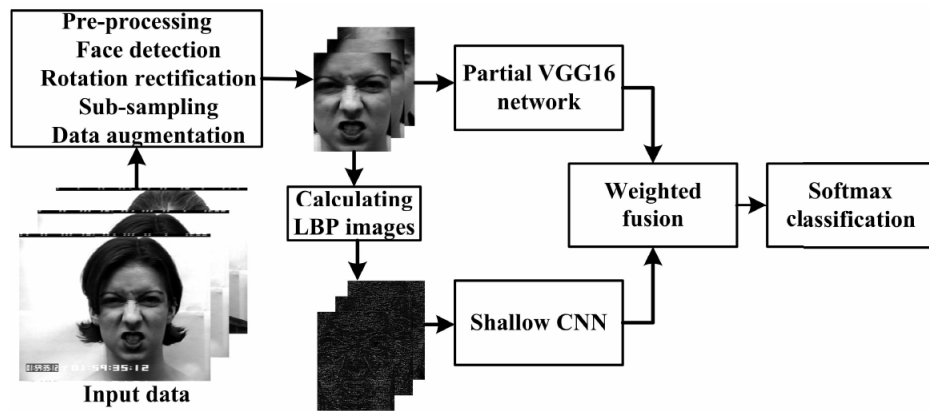
<i>NEUTRAL</i>	AU 9	AU 10	AU 12	AU 20
				
Lips relaxed and closed.	The infraorbital triangle and center of the upper lip are pulled upwards. Nasal root wrinkling is present.	The infraorbital triangle is pushed upwards. Upper lip is raised. Causes angular bend in shape of upper lip. Nasal root wrinkle is absent.	Lip corners are pulled obliquely.	The lips and the lower portion of the nasolabial furrow are pulled back laterally. The mouth is elongated.
AU15	AU 17	AU 25	AU 26	AU 27
				
The corners of the lips are pulled down.	The chin boss is pushed upwards.	Lips are relaxed and parted.	Lips are relaxed and parted; mandible is lowered.	Mouth stretched open and the mandible pulled downwards.
AU 23+24	AU 9+17	AU9+25	AU9+17+23+24	AU10+17
				
Lips tightened, narrowed, and pressed together.				
AU 10+25	AU 10+15+17	AU 12+25	AU12+26	AU 15+17
				
AU 17+23+24	AU 20+25			
				

(b) Lower face AUs.

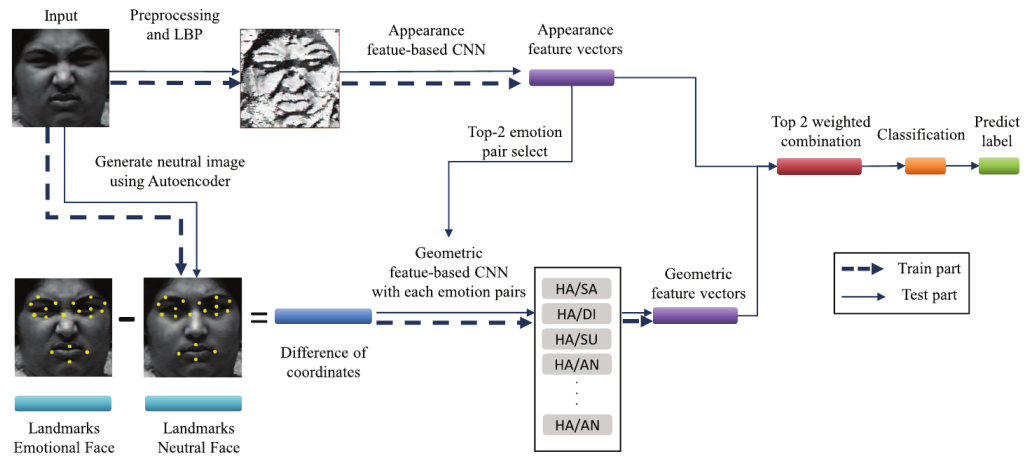
Figure 1.1: Examples of face action units (AUs). Taken from [1].



(a) I^2 CNN: One channel architecture. Taken from [25].



(b) WMDNN: Two channels architecture. Taken from [5].



(c) Hierarchical DNN: Two channels architecture. Taken from [6].

Figure 1.2: Examples of CNN-based FER systems architectures proposed in literature.

2 Data sets for FER systems

Data sets suitable for training FER systems are composed of several images that share characteristics like resolution, color, etc. Usually, these data sets must face problems such as [14]:

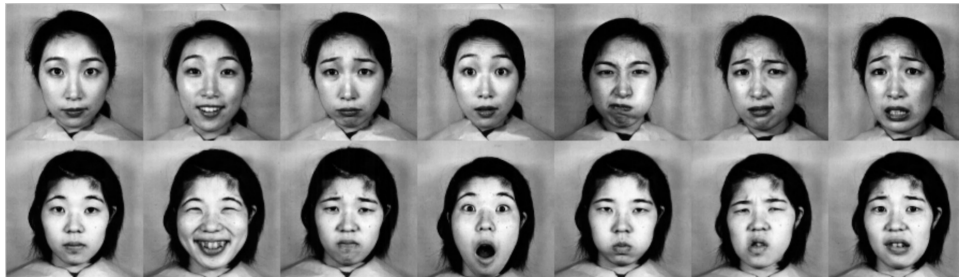
- Emotions are expressed at different intensities for each subject.
- There is a loss of authenticity when the subjects are aware that they are being photographed.
- Subject's spontaneous expressions may be limited or inhibited by the environment.

Among all the different data sets available, the most popular in literature are the Cohn-Kanade data set, proposed in 1998 [1] and extended as CK+ in 2010 [26], and the Japanese Female Facial Expression (JAFFE) data set, proposed in [23] and [2].

The Cohn-Kanade data set (CK+) is composed of 500 images from 100 subjects with ages ranging from 18 to 30 years, and ethnicity is distributed as 15% African-Americans and 3% Asians and Latino-Americans. 65% images are of women, and the data set includes only posed expressions. Images were taken using two cameras: one directly in front of the subject and the other positioned 30 degrees to the subject's right. However, the data set contains only the images taken from the frontal camera. Some images from the CK and CK+ data sets are shown in figure 1.3a.



(a) Example images from CK (1st row) and CK+ (2nd row). Taken from [26].



(b) Example images from JAFFE. Taken from [2].

Figure 1.3: Examples of data sets used in FER systems.

On the other hand, the Japanese Female Facial Expression (JAFFE) data set is composed of 219 images of 7 facial expressions (6 basic facial expressions + 1 neutral) with only posed expressions from 10 Japanese female models [23]. The photos have been taken under strictly controlled conditions with similar lighting by using the acquisition setup presented in figure 1.4. Models have tied their hair away from their faces to avoid hiding their expressions. Figure 1.3b shows some images from the JAFFE data set.

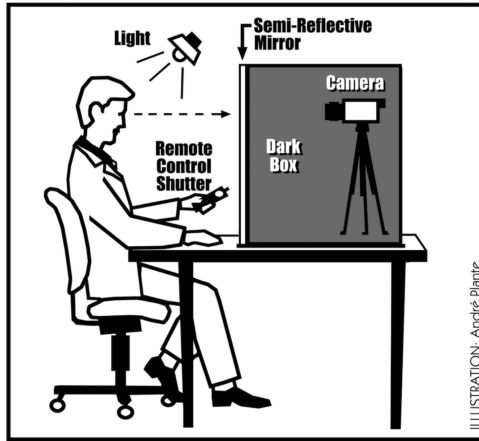


Figure 1.4: Acquisition setup used for JAFFE. Taken from [2].

3 Basic data set preprocessing for FER systems

Data set preprocessing can improve the FER system's accuracy by eliminating irrelevant information such as the background, and adjusting parameters such as the input size, resolution, etc. Figure 1.5 summarizes the steps involved in basic data set preprocessing.

The original image has a size of 256×256 pixels and comes directly from the data set. Faces are commonly detected by using algorithms such as the Adaboost learning algorithm, proposed by Viola and Jones in 2004 [27]. In this work, the landmarks released by Davis King in [28] and first published in [4], are employed (see figure 1.6). These landmarks are read using a C++ toolkit for ML algorithms known as Dlib [29].

The Dlib methods used are *get_frontal_face_detector()* and *shape_predictor(landmarks)*, which are both used in Anas Khayata's algorithm [3]. For instance, the eye detection algorithm uses the points 36 – 42 for the right eye and the points 42 – 48 for the left eye.

Next, the rotation angle is calculated by using a line between the eyes, while a vertical line traversing the image is used as a reference (see figure 1.5). Rotation makes use of the functions *getRotationMatrix2D* and *warpAffine* from the OpenCV library [30].

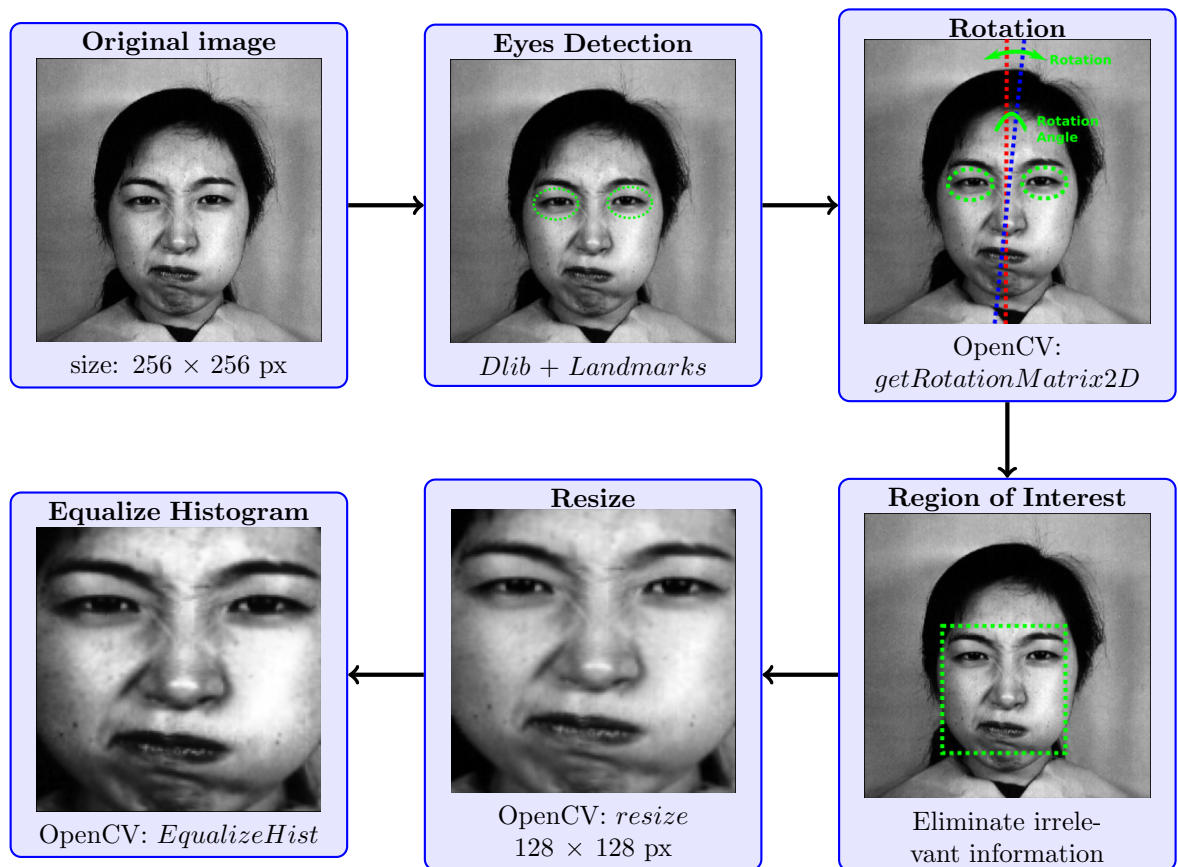


Figure 1.5: JAFFE Pre-processing for FER. Adapted from [3].

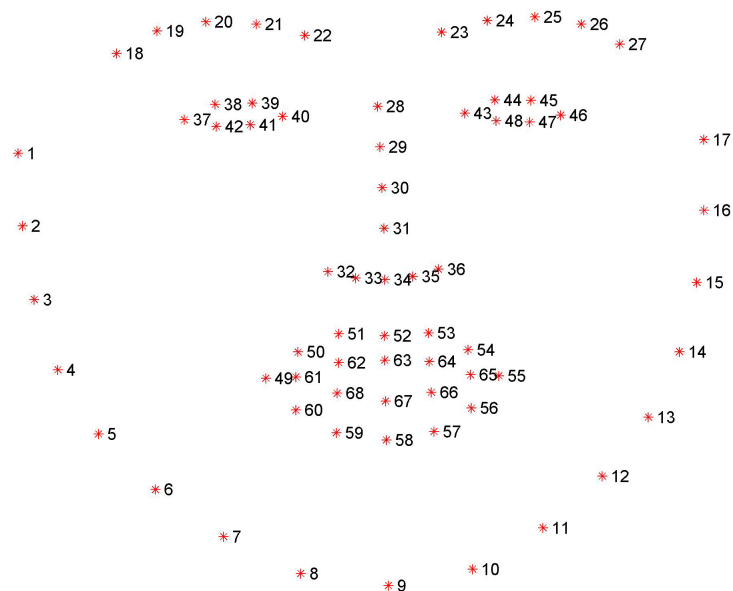


Figure 1.6: Facial Point Annotations. Taken from [4].

After that, the Region of Interest (ROI) is cropped from the image, eliminating background, and non-relevant parts such as the hair and the neck. The image is then resized to 128×128 pixels. Lastly, the image contrast is improved by Histogram Equalization using the OpenCV `equalizeHist()` function. This implementation can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/00_jaffe_pre-traitement/00_dlib_lbp_comparaison/mes_fonctions.py.

4 Extended data set preprocessing for FER systems

4.1 Local Binary Pattern

The visual descriptor known as Local Binary Pattern (LBP) is frequently used in FER systems [5], [6], due to the enhancement of textures associated with the action units (AU). To compute the LBP codes, a pixel from the grayscale image is compared to the pixels around it in a 3×3 array. The encoding process is summarized by equations 1.1 and 1.2, where i_c and i_p represent the grayscale image values from the center and its neighbors pixels, respectively, and N is the number of neighbors' pixels. The encoding process ends with a decimal value used to build the new LBP image, as shown in figure 1.7.

$$LBP = \sum_{n=1}^N s(i_p - i_c) \times 2^n \quad (1.1)$$

$$s(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases} \quad (1.2)$$

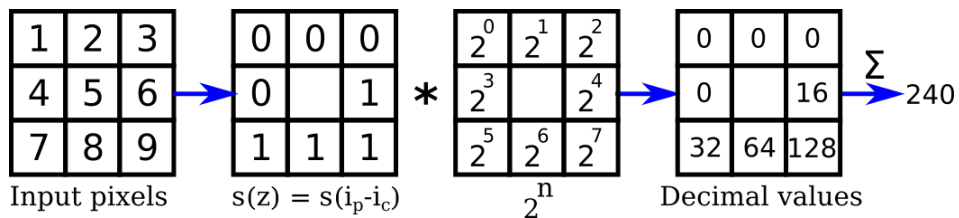


Figure 1.7: LBP encoding example. Adapted from [5], [6].

Implementation of LBP in this work was made by employing the Scikit-image library [31]. In particular, the *local_binary_pattern* function from *skimage.feature* was used. This function provides five methods for determining the pattern as described in [32]:

- **“default”**: Rotation-variant LBP.
- **“ror”**: Rotation-invariant LBP.
- **“uniform”**: LBP using improved rotation-invariance and uniform patterns.
- **“nri_unifor”**: LBP with uniform patterns but non-rotation-invariant.
- **“var”**: LBP based on rotation-invariant variance measures of the contrast of local image texture.

Figure 1.8 shows the different LBP methods that were applied to the same JAFFE image. Implementation can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/00_jaffe_pre-traitemment/00_dlib_lbp_comparaison/mes_fonctions.py.

As the smoothest result was obtained with the “var” method, this was selected for applying LBP to the entire JAFFE data set. Finally, the LBP image is resized to 128×128 pixels using *OpenCV.resize* with LANCZOS4 as the interpolation method.

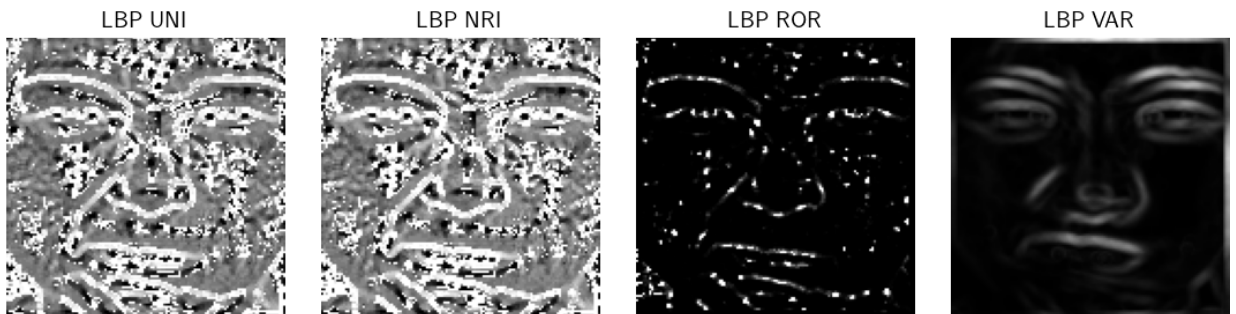


Figure 1.8: LBP methods comparison with JAFFE.

4.2 Data Augmentation

Training CNNs requires a large number of images to get acceptable precision and avoid overfitting. As the JAFFE data set size is about 256 images, it is necessary to increase the number of training samples by means of techniques like data augmentation (DA).

In DA, each training sample is subject to transformations such as rotation, flip, etc.; to create new training data. In this work, a Python library for data augmentation named Albumentation was used [33]. 15 transformations, including shift, scale, rotation, flip, and some combinations, were applied to images of 6 emotions as reported in the literature [5], [6].

After this process, the number of training images increased to 2640. Figure 1.9 depicts some DA examples over JAFFE resized to 64×64 , and the implementation can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/00_jaffe_pre-traitement/07_dlib_var_red_aug15/00_dlib_var_red_aug15.py.

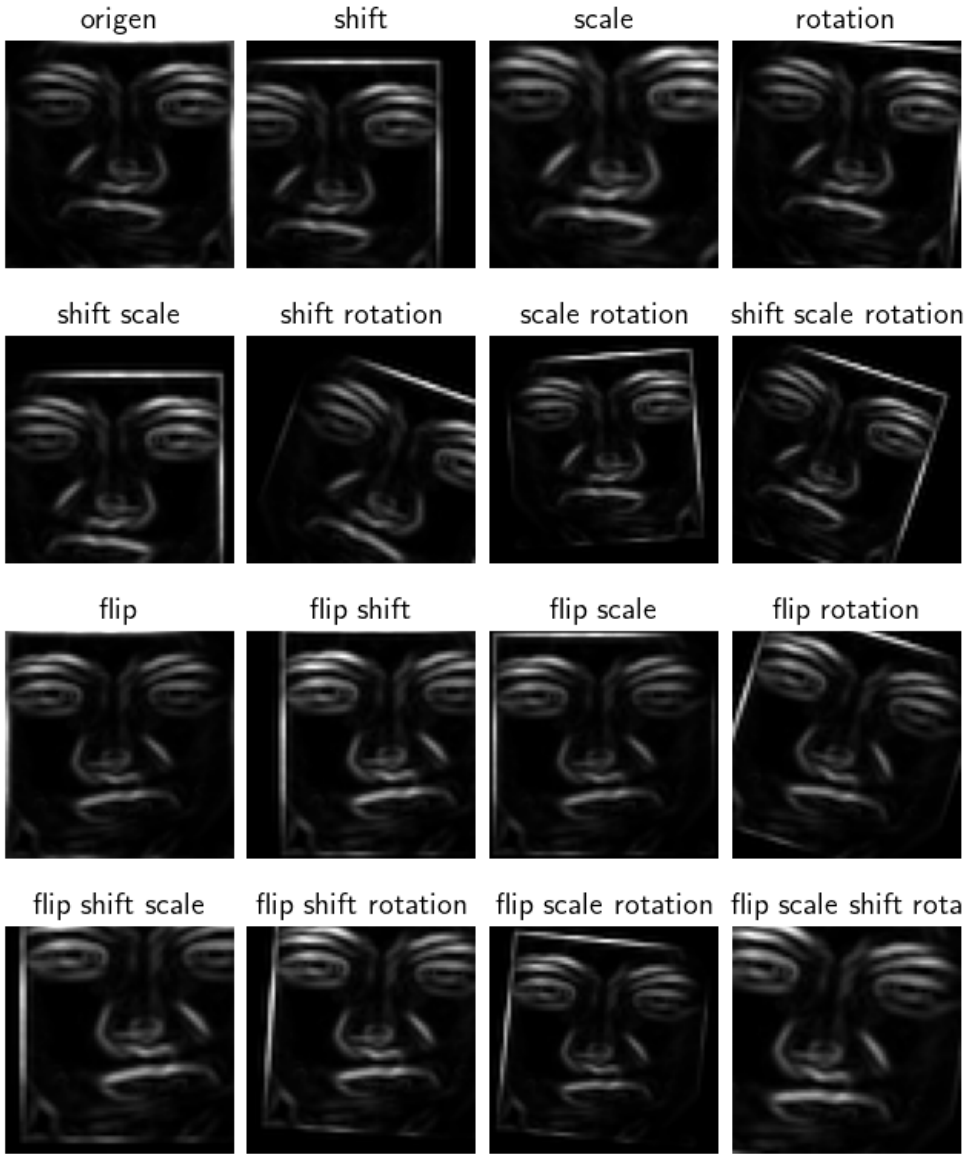


Figure 1.9: Examples of data augmentation with JAFFE.

5 Designing a CNN-based FER system

The architectures of some CNN-based FER systems proposed in the literature [25], [5], [6], [17] were presented in subsection 1.1. The CNN models designed in those works are summarized in table 1.1, as well as the accuracy obtained with the JAFFE data set.

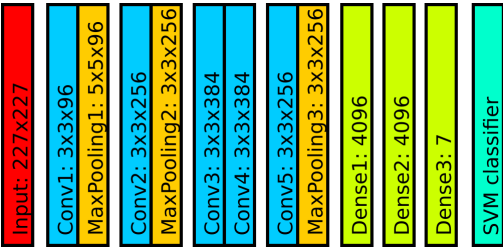
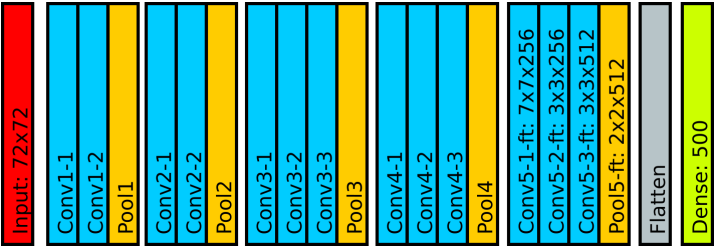
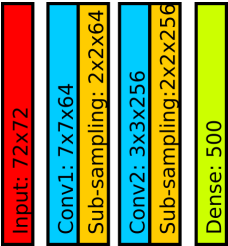
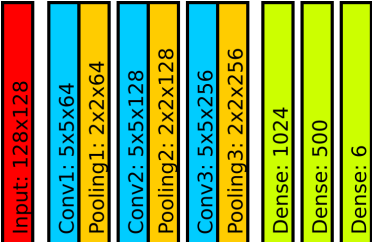
Ref.	Model	Accuracy	Architecture
[25]	I^2CNN for grayscale images	75.28%	
[5]	$WMDNN$ Channel 1: Partial VGG16 for grayscale images	90.86%	
[5]	$WMDNN$ Channel 2: Shal- low CNN for LBP images	88.33%	
[6]	Appearance feature- based CNN for LBP images	89.33%	

Table 1.1: Architectures of some CNN-based FER systems and their accuracy for JAFFE.

On the other hand, the performance of CNN inference on hardware-software architectures is affected by the Logic Resources available, the operations involved, and the model size. Thus, the most suitable option for an embedded system is a single-channel CNN-based FER system.

The CNN's design is an iterative process that depends on previously acquired knowledge about the input data and the model's accuracy. Moreover, expertise on CNNs could limit the design exploration space (DES) to tuning the most relevant parameters. Besides, some known tips are:

- Start with a convolutional layer with a few filters and increase the filters in the following layers. The number of filters should be multiples of two.
- Use a subsampling layer such as *AveragePooling* or *MaxPooling* after each convolutional layer to decrease the size of the feature maps.
- The number of neurons in the last layer (e.g., Dense) should be equal to the number of classes. Remember that for the FER system, each class corresponds to an emotion.
- Compare the performance of different optimizers (e.g., SGD, Adam) using metrics such as accuracy, loss or graphics like learning curves.
- Adjust the learning rate (lr , α) to improve the training process. Some of the recommended values are $\alpha = 0.001$ and $\alpha = 0.0005$.
- Tune the batch size (bs) and the number of epochs (a.k.a. *époques*) to avoid underfitting and overfitting.

This work employed the JAFFE data set with 6 and 7 emotions, processed with the Local Binary Pattern (LBP) using the VAR method and data augmentation with Albumentation, as well as TensorFlow to train the CNN models. TensorFlow is an open-source framework for developing machine learning systems [19]. To evaluate the model's performance, the learning curves for accuracy (a.k.a. exactitude) and loss (a.k.a. *perte*) were analyzed. Hence, three diagnostics are possible:

- **Underfitting:** The model doesn't reach a lower error value, and the training loss doesn't decrease no matter the number of epochs. Another scenario is that the training loss is decreasing but the training is halted, thus it is needed to increase the number of epochs until the curve gets stable.

- **Overfitting:** The model learned the training data set but fails to generalize, as a consequence, it presents high accuracy with the training data set and low accuracy with the validation data set. The validation loss curve could decrease in the first epochs and then start to increase, creating a gap between the validation and training loss curves.
- **Good fit:** Can be identified when the curves for the validation loss and the training loss decrease to an acceptable value, while keeping a small gap between the curves. Here, it is important to observe that increasing the number of epochs without control can result in overfitting.

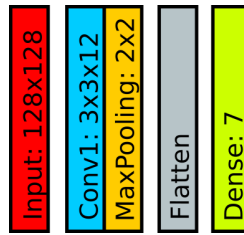
Additionally, the ML framework typically provide techniques to fine-tune the training. For instance, `EarlyStopping` allows to end the training when a metric (e.g., validation loss) no longer improves [34], whereas, `ReduceLROnPlateau` decreases the learning rate value when a metric improvement halts [35].

In the following subsections, some models tested are presented, as well as the model selected to develop the work goals. The ML framework employed was TensorFlow [19]. TensorFlow is an open source project currently used by several companies, including AMD, ARM, Google, Intel, Lenovo, Nvidia, and Texas Instruments, among others. More than 60 tests were performed with 3 to 10 repetitions. In addition, the source code and the training reports, including the model architectures and the learning curves, can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/00_sw/01_etu_mod_jaffe/01_MX_fp.

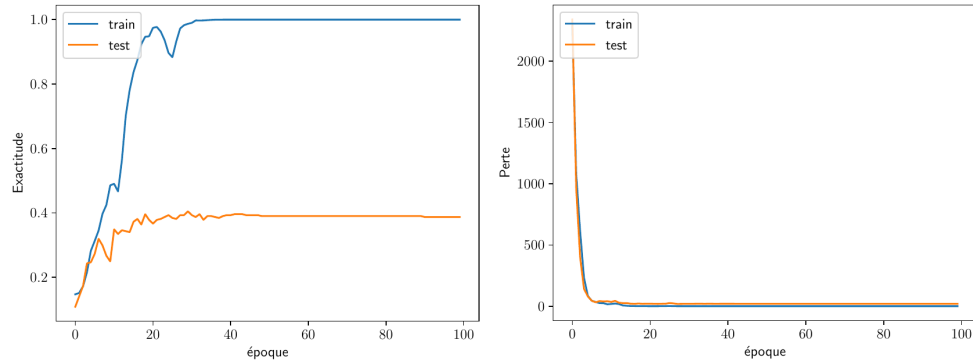
5.1 Testing model M1

The architecture of model M1 (see figure 1.10) was adapted from the example “Training a neural network on MNIST with Keras” found in https://www.tensorflow.org/datasets/keras_example. It was chosen as the starting point because, for the MNIST data set, it reached an accuracy above 90% with just a few layers.

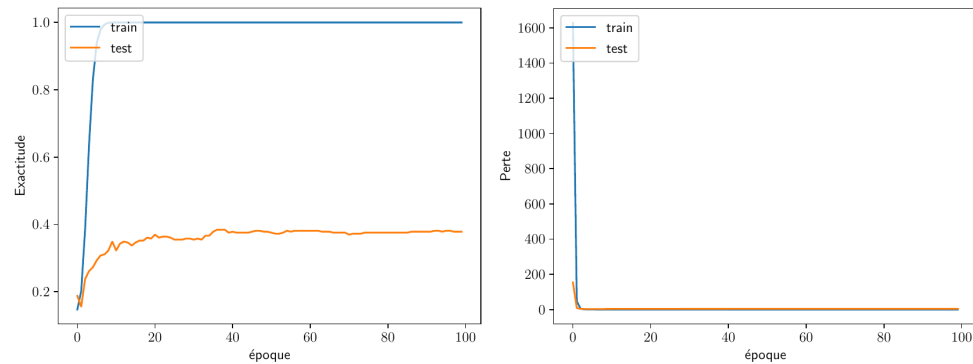
The M1 input for FER came from the JAFFE data set with samples of 128×128 *px*. The parameters under test were the batch size (bs), the learning rate (lr) and the number of filters in the convolutional layers, among others.



(a) Original architecture.



(b) $bs = 256$, $lr = 0.001$.



(c) $bs = 128$, $lr = 0.001$, with 128 filters.

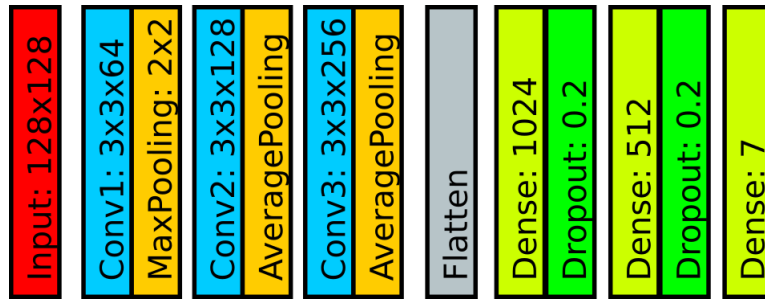
Figure 1.10: Tests with the M1 model.

According to the accuracy and loss learning curves presented in figure 1.10, the model is underfitting with less than 50% of accuracy for the validation data set. These tests are located in https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/01_etu_mod_jaffe/01_MX_fp/06_entrainement_dlib128_aug15_20210219.pdf.

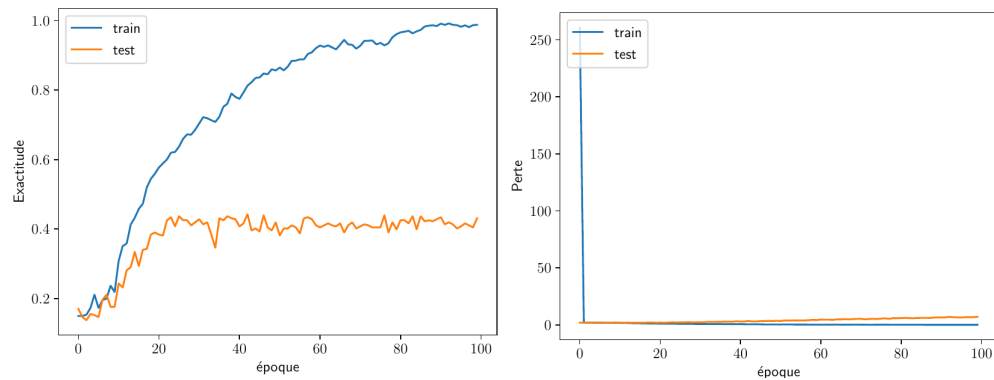
5.2 Testing model M2

The architecture of model M2 (see figure 1.11) is based on the partial VGG16 proposed for the WMDNN FER system [5] shown in table 1.1.

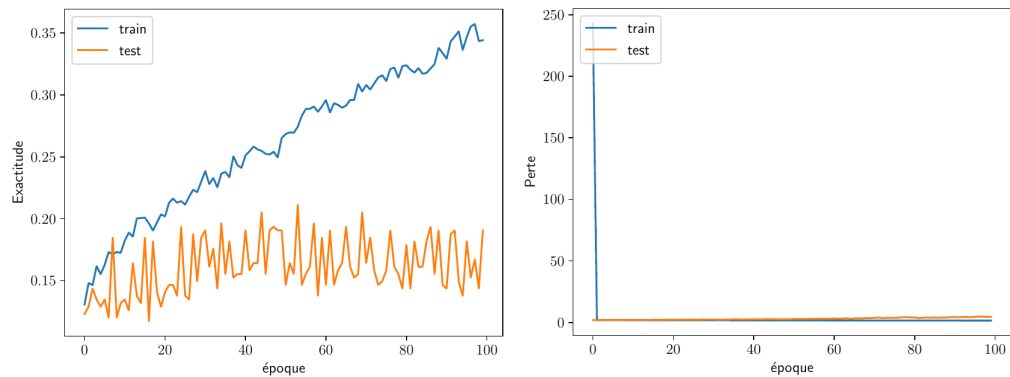
The model was tested with and without *Dropout* varying the batch size, the learning rate, and using *MaxPooling* instead of *AveragePooling*. The learning curves show instability, and the accuracy is under 50%. The training reports for this model can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/01_etu_mod_jaffe/01_MX_fp/01_entrainement_dlib128_aug15_20210217.pdf.



(a) Original architecture.



(b) $bs = 256$, $lr = 0.001$

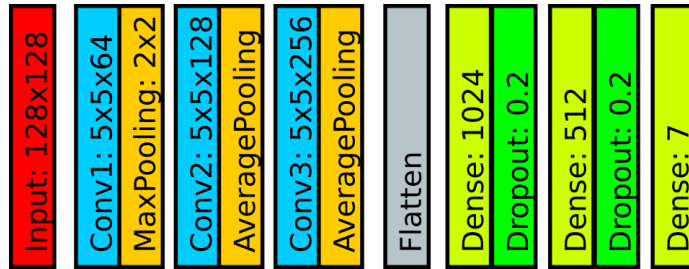


(c) $bs = 128$, $lr = 0.001$, without dropout and MaxPooling instead of AveragePooling.

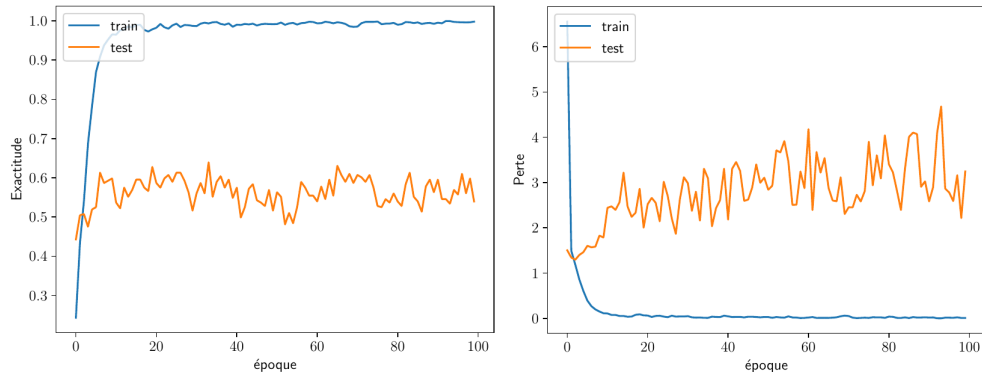
Figure 1.11: Tests with the M2 model.

5.3 Testing model M3

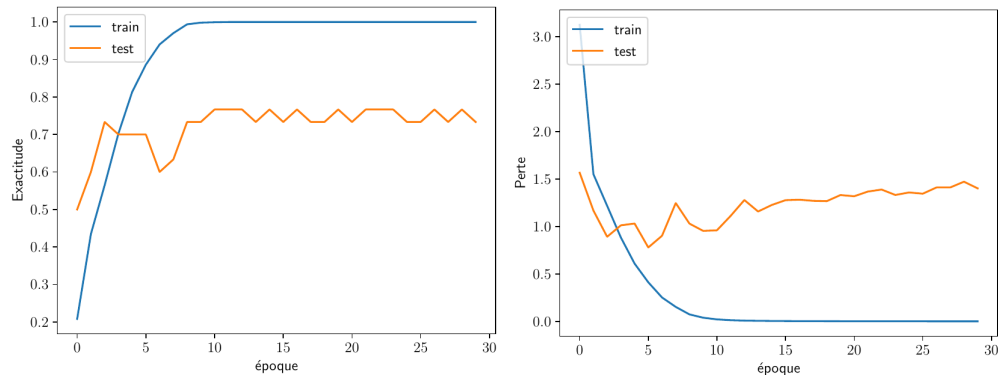
The architecture of model M3 (see figure 1.12) is based on model M2 but the kernel size for the convolutional layers is increased from 3×3 to 5×5 . The tests made with M3 include varying the batch size, the learning rate, with and without *Dropout* and using *MaxPooling* instead of *AveragePooling*. The learning curves obtained show accuracy improvements up to 80% after 100 epochs (époques), but the model tends to overfit. These tests are located in https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/01_etu_mod_jaffe/01_MX_fp/17_entrainement_dlib_var128_aug15_20210219.pdf.



(a) Original architecture.



(b) $bs = 64$, $lr = 0.001$



(c) $bs = 64$, $lr = 0.001$, without Dropout, Dense(1024), Dense(512) and MaxPooling instead of AveragePooling.

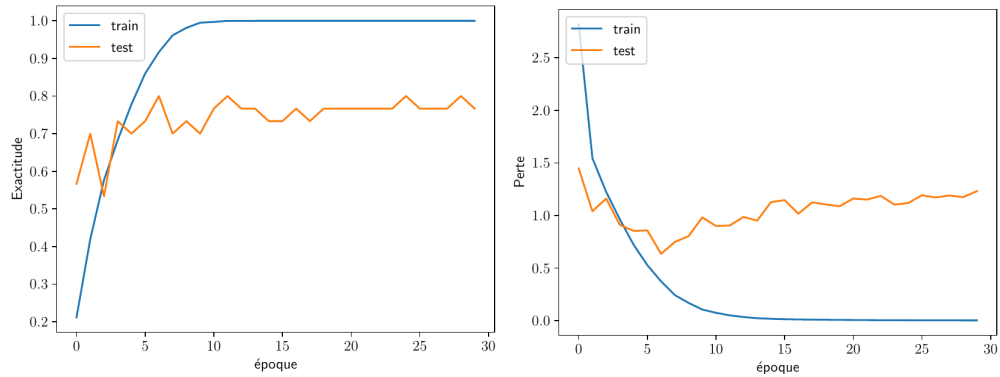
Figure 1.12: Tests with the M3 model.

5.4 Testing model M4

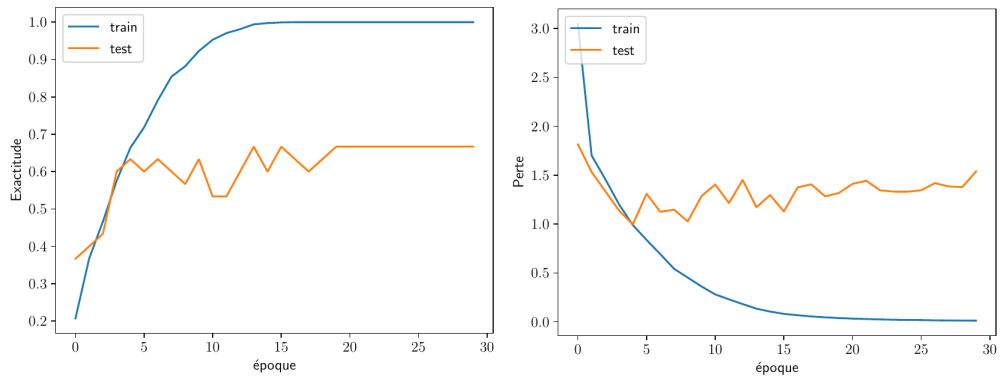
The architecture of model M4 (see figure 1.13) is based on model M3 but includes only 1 *Dense* layer to avoid overfitting. Besides, the input size is changed to 64×64 and the number of epochs is reduced from 100 to 30, because the accuracy and loss barely improve after epoch 15. The model was tested by varying the batch size, the learning rate, and the number of filters per *Convolutional* layer among other parameters. The learning curves show a validation accuracy of around 70% but the gap between the training and the validation loss indicates that overfitting is still present. The test reports can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/01_etu_mod_jaffe/01_MX_fp/42_entrainement_dlib_var64_aug15_20210302.pdf.



(a) Original architecture



(b) $bs = 64$, $lr = 0.0001$



(c) $bs = 64$, $lr = 0.0001$, with Conv of 32, 64 and 128 filtres, and kernel of (5,5).

Figure 1.13: Tests with the M4 model.

5.5 Testing model M6

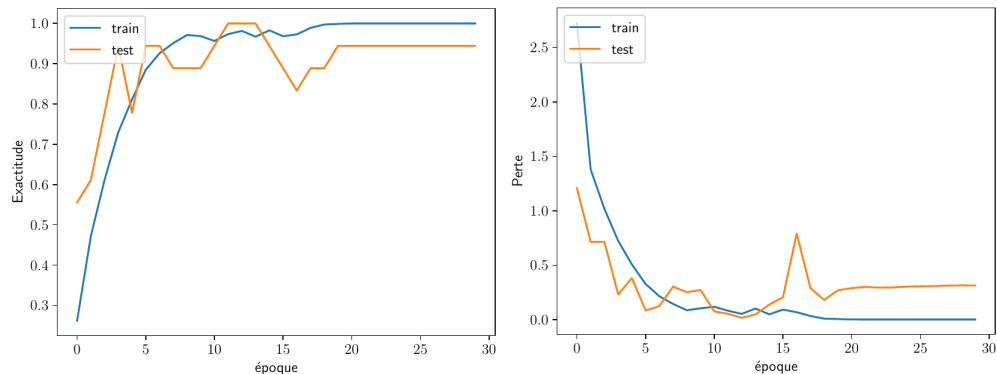
Model M5 is based on model M4, but the stride parameter is replaced for the *Convolutional* and the *MaxPooling* layers from (2, 2) to (1, 1). Test reports for this model are located in

https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/01_etu_mod_jaffe/01_MX_fp/60_entrainement_M5_jaffe_dlib_var64_aug15_20210313.pdf.

The architecture of model M6 (see figure 1.14) is based on model M5 but changes the number of filters in the *Convolutional* layer to 32, 64, and 128. In addition, the *Dense* layer includes six neurons instead of seven, thus training is made with the data set including 6 emotions, as found in the literature. The learning curves show the validation accuracy stabilizing around of 90% and the validation loss around 0.4. It is worth mentioning that the gap between the training and the validation loss is approximately 0.3, which indicates an acceptable fit. As the model M6 is a single-channel CNN that achieves acceptable performance, this will be the CNN-based FER system proposed in this work. The training reports for model M6 can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/02_M6_jaffe/00_M6_jaffe_dlib_var64_aug15/00_entrainement_M6_jaffe_dlib_var64_aug15_6emo_20210324.pdf The python script used is also available in the appendix B as listing B.1.



(a) Original architecture



(b) $bs = 32$, $lr = 0.001$

Figure 1.14: Tests with the M6 model.

6 Discussion

The recent use of CNN models for FER systems has been widely adopted in the state of the art, mainly because of the performance that can be attained. However, because there is not a clear agreement on what architecture should be used for the FER system, it remains an intriguing topic for research.

Moreover, the applications for FER systems have different requirements with different relevance according to the specific problem. For instance, latency and accuracy are crucial for automobile safety, while computational complexity could be more relevant for embedded systems with human-computer interaction capabilities.

On the other hand, although there are several algorithms that could be used to enhance the data set (e.g., the Canny Edge detector), the stages implemented in this work for the basic and extended preprocessing were taken from the literature, making the results comparable with similar works.

Besides, this work trained a single-channel CNN with a small data set, which led to learning curves for accuracy and loss that had ripple effects with some outliers. This behavior could cause techniques like EarlyStop and ReduceLRonPlateau to randomly stop the training or change the learning rate before getting to the optimal values.

Furthermore, because the work aims to run CNNs inference in a hardware-software architecture, the design process of a CNN-based FER system involves a trade-off between performance and the computational resources available in the target platform.

7 Conclusion

This chapter briefly presented the basis of FER systems, from their traditional architectures to those integrating ML algorithms. Furthermore, due to the inherent complexity of FER systems, it is advised to follow the literature along with the data sets used, the basic preprocessing stages (e.g., eliminating background, face rotation, histogram equalization, etc.), the LBP and the DA. Hence, it is worth mentioning that “the input data could be as important as the system architecture”.

In addition, the design process of a CNN-based FER system was described. From the several tests made, the relevance of each parameter is studied, as are the relationships between overfitting/underfitting and the number of layers, samples, epochs, etc.

In addition, as mentioned before, designing a FER system is an observation process with several iterations. Where analyzing the learning curves and not only the final values of metrics is essential. Lastly, a single-channel CNN-based FER system trained with TensorFlow and the JAFFE data set that achieves an accuracy of above 90% is proposed.

Hardware Neural Networks (HNN)

In chapter 1 model M6 was proposed as a single-channel CNN-based FER system. This model was trained using the JAFFE data set [23], [2] and TensorFlow, obtaining parameters represented by floating-point numbers. This chapter will employ model M6 to design and propose a hardware-software architecture able to execute the inference of a FER CNN model. This architecture will be an alternative to traditional and expensive platforms such as GPUs and high-performance systems.

An outlook of AI-enabled IoT devices is presented in [21], along with two techniques of model compression: pruning and quantization. Pruning removes connections, neurons, or layers from the network to reduce the memory and computational resources needed [36]. However, it could decrease the accuracy. Whereas quantization typically keeps the model's architecture but changes the numeric representation of the parameters, which could also affect the accuracy. Hence, due to M6's size, the less aggressive compression technique will be used, which is quantization.

The chapter starts by introducing the quantization library tflite from TensorFlow. Tflite allows representing neural network parameters as integer numbers. As a result, they can be imported to hardware platforms such as FPGAs without losing too much precision. Besides, in section 2 the development platform employed is introduced. Then, the configuration of a basic SoC and the cores capable of performing tflite operations like the *Convolution* and the *MultiplyByQuantizedMultiplier* are described in section 3. Following, section 4 shows the integration of the aforementioned cores into the FER SoC, which is appropriate for running the inference of model M6. The address assigned to each core, the number of resources used, and the power consumption are also specified. The chapter provides links to the Vivado and Vitis projects, making the results reproducible. Lastly, the discussion and conclusions can be found in sections 5 and 6.

1 Quantization Aware Training

Running the inference of CNN models in embedded systems is a task that could entail complex logic resources and a reasonable amount of memory. This is due to the model's size and the need for storing the layers' outputs, usually in floating-point representation. Besides, the Quantization Aware Training technique transforms a floating-point TensorFlow model into an integer model with similar performance. It was proposed in [13], and is now implemented in the TensorFlow package named *model_optimization*, which is part of the TensorFlow Lite source code.

For instance, Figure 2.1 shows how a convolutional layer from the M6 model is quantized. After all the model layers have been quantized, the model's inference will only involve integer-arithmetic operations.

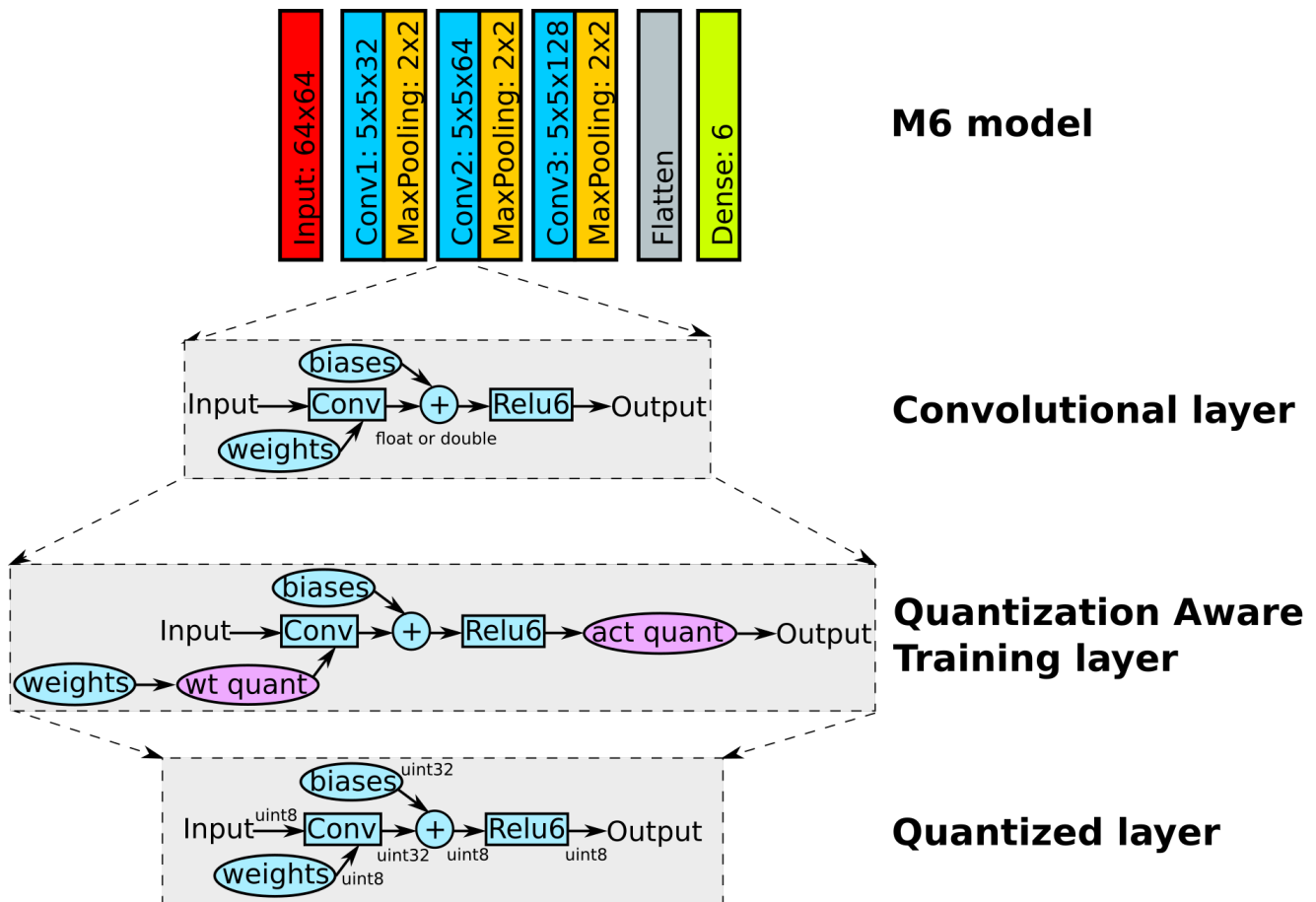


Figure 2.1: Tflite Quantization Aware Training Example.

On the other hand, the impact of the quantization process on the model's accuracy depends on factors such as the layers and the training parameters, including the data set. However, there are some models whose accuracy is not so different after quantization. Table 2.1 presents some models suitable for quantization.

Before applying quantization to a model, it is necessary to check the supported operations. Those can be found at the TensorFlow Lite 8-bit quantization specification (see [37]), as well as the expected range of the numeric values. Table 2.2 describes the specification for the layers employed by model M6.

Model	Floating-point baseline model	Quantized Aware Training Model	Post-training full integer quantized Model
MobileNet v1 1.0 224	71.03%	71.06%	69.57%
MobileNet v2 1.0 224	70.77%	70.01%	70.20%
ResNet v1 50	76.30%	76.10%	75.95%

Table 2.1: Quantized models with comparable accuracy to floating point. Adapted from [20].

Layer	Inputs/Outputs	Data type	Range
Conv_2D	Input 0:	int8.	$[-128, 127]$.
	Input 1 (Weight):	int8.	$[-127, 127]$.
	Input 2 (Bias):	int32.	$[int32_{min}, int32_{max}]$.
	Output 0:	int8.	$[-128, 127]$.
Fully Connected	Input 0:	int8.	$[-128, 127]$.
	Input 1 (Weight):	int8.	$[-127, 127]$.
	Input 2 (Bias):	int32.	$[int32_{min}, int32_{max}]$.
	Output 0:	int8.	$[-128, 127]$.
Max_	Input 0:	int8.	$[-128, 127]$.
Pool_2D	Output 0:	int8.	$[-128, 127]$.

Table 2.2: Specification for the quantized layers used in M6. Adapted from [37].

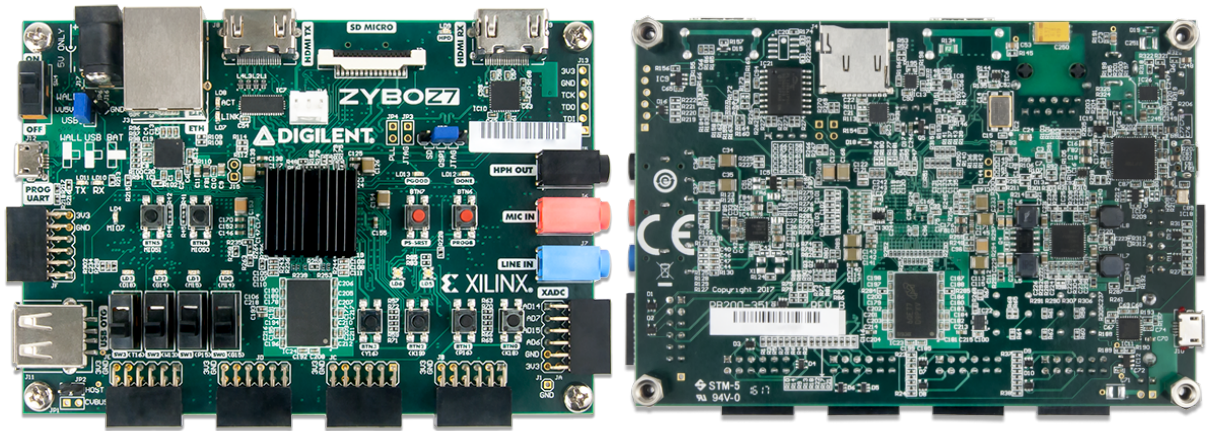
As all the M6 layers were supported by TensorFlow Lite, quantization was applied to the model, and performance was evaluated through the metrics displayed in table 2.3. Although the accuracy decreased to 83.3% and the loss increased to 0.36, having integer-only inference, is a tradeoff that makes the hardware acceleration more feasible. This implementation can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/02_M6_jaffe/01_M6_jaffe_tflite/entrainment.log.

Model Representation	Accuracy	Loss	Parameters	Size MB
Floating Point	94.44%	0.12	306.182	1.17
Quantized	83.33%	0.36	306.652	0.30

Table 2.3: Metrics for model M6 in float point and after quantization.

2 Hardware Platform

A hardware-software co-design allows the execution of an algorithm by using designs that share hardware and software components on specialized platforms. These platforms are mainly composed of microcontrollers or embedded processors with specific IP cores implemented in FPGAs. FPGAs are programmable integrated circuits, considered one of the best tools for the design of digital circuits due to their capability of being dynamically reconfigured and their affordable prices [38]. Usually, FPGAs are sold with development boards that come with the JTAG programmer, Pmods and input and output peripherals (e.g., HDMI, USB, DDR memory, etc.). Digilent's Zybo-Z7 [8] is employed in this work (see Figure 2.2). The academic price for this board is around *USD*\$262, which is not expensive for this kind of platform. The board integrates the XC7Z020-1CLG400C Zynq FPGA from Xilinx.



(a) Top.

(b) Bottom.

Figure 2.2: Zybo-Z7 Development board. Taken from [7].

The Zynq family incorporates a processing system (PS) with traditional programmable logic (PL) in the same chip, as shown in figure 2.3. Also, a more detailed representation can be found at C.2. The PS is a multi-core ARM 32-bit processor running at 660 MHz interconnected to the PL by an AMBA interface, such as Axi4-Lite. Hence, it is possible to design a System on Chip (SoC) with both hard-core and soft-core processors.

The PL includes configurable basic elements for implementing digital designs as complex as soft core processors. Some of these elements include:

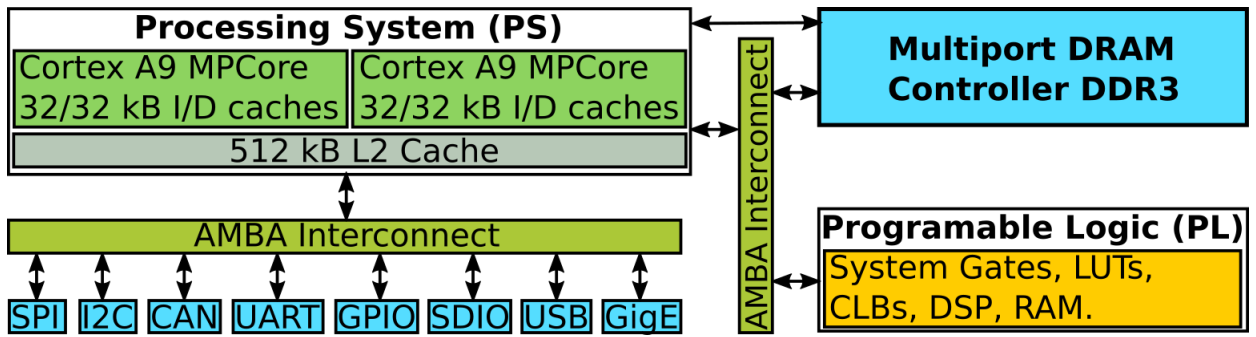
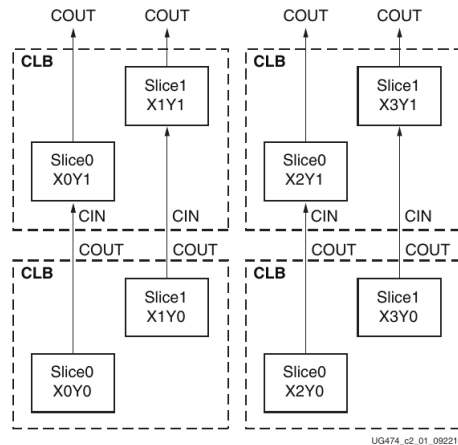


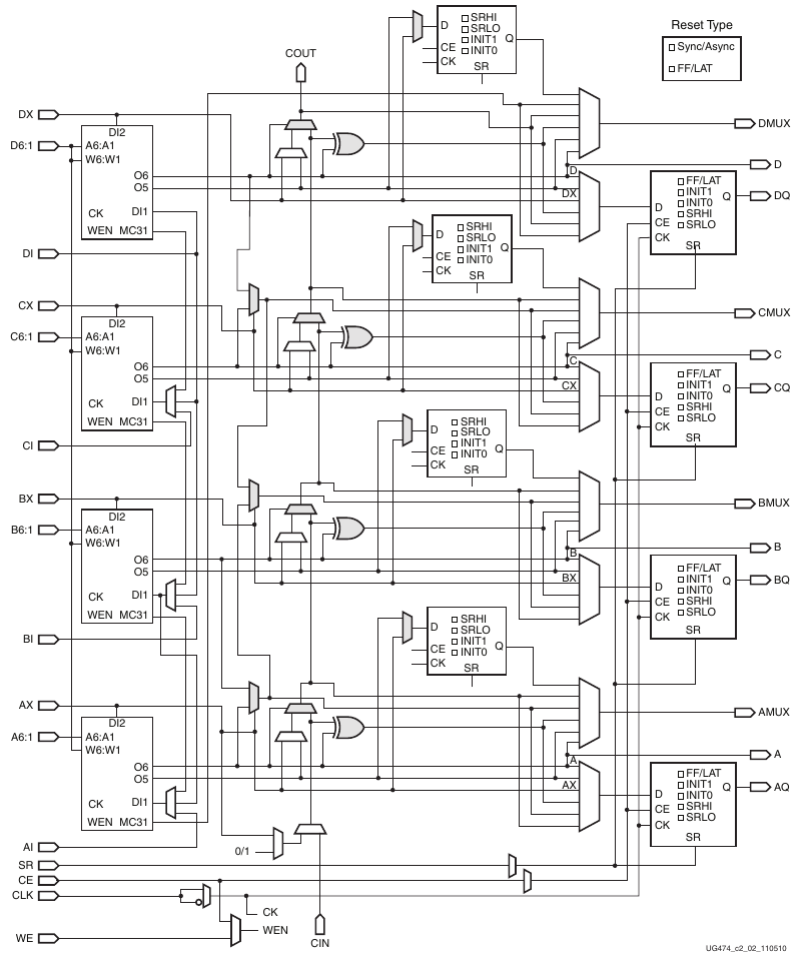
Figure 2.3: Zynq APSoC Architecture Simplified. Adapted from [8].

- **Configurable Logic Block (CLB):** One CLB has 8 LUTs, 16 Flip-flops, 2 arithmetic and carry chains, 256 bits of distributed ram and 128 bits of shift registers [9]. As shown in figure 2.4, these resources are grouped in 2 slices. Slices inside a CLB don't share connections and can be slices with memory (SLICEM) or without memory (SLICEL).
- **Block RAM (BRAM):** According to [10] a block RAM can store up to 32 kbits and has two independent access ports. Thus, data can be written to and read from either port. Write and read operations are synchronous, and hence they need a clock's edge. The structure of a BRAM is shown in figure 2.5. BRAMs can be arranged to provide the following configurations: $64k \times 1$, $32k \times 1$, $16k \times 2$, $8k \times 4$, $4k \times 9$, $2k \times 18$. BRAMs can be used to store the layer's parameters.
- **DSP48 Block:** This resource can be used to implement the addition, multiplication, and multiply and add operations. As seen in figure 2.6, this block is made up of a 25×18 two's complement multiplier, a 48-bit accumulator, and a Single-Instruction-Multiple-Data (SIMD) arithmetic unit, among other elements [11]. The use of these elements in a specific IP core can be inferred by the design tool (i.e., Vivado). However, there are some primitives that allow you to define when they should be used.

Table 2.4 summarizes some of the resources available on the XC7Z020-1CLG400C FPGA in the Zybo-Z7 board employed.



(a) CLB structure.



(b) SLICEM.

Figure 2.4: PL architecture. Taken from [9].

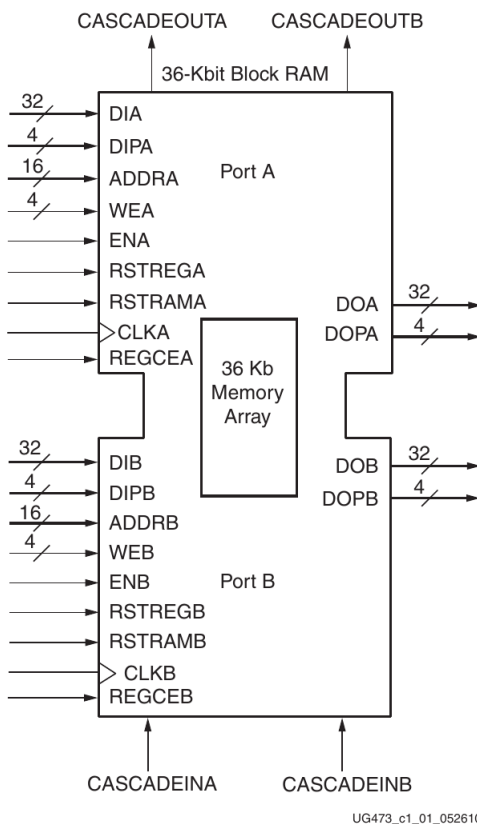


Figure 2.5: BRAM structure. Taken from [10].

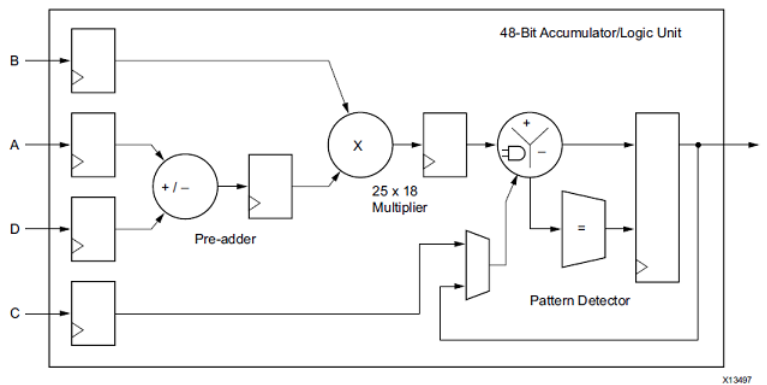


Figure 2.6: Basic DSP48 Slice. Taken from [11].

Logic Resource	Quantity
Logic slices	13,000
6-input LUTs	53,200
Block RAM	630 <i>kB</i>
DSP	220

Table 2.4: XC7Z020-1CLG400C Specification. Taken from [7].

3 Hardware-software Development

3.1 Base SoC Design

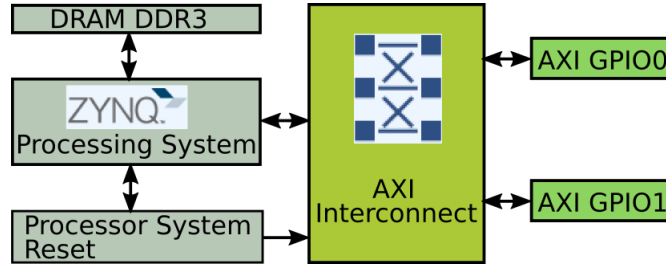


Figure 2.7: Basic ARM SoC.

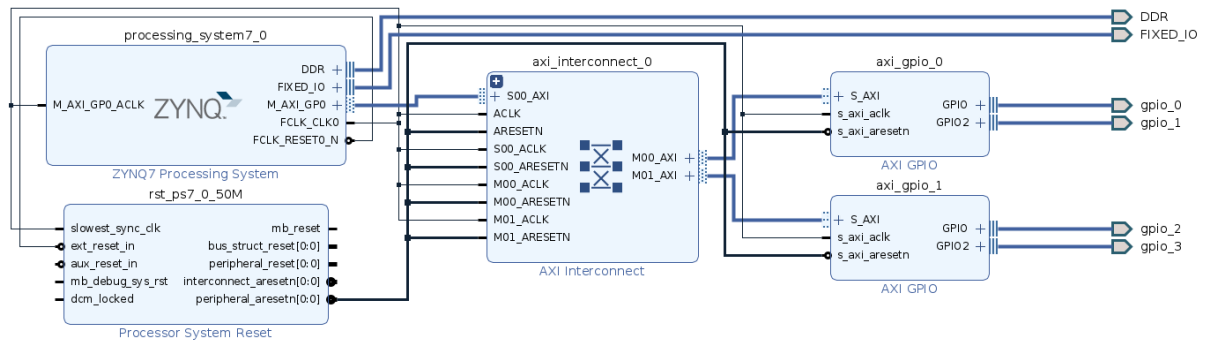


Figure 2.8: Vivado Block Design for the Basic ARM SoC.

Figure 2.7 shows a basic System on Chip (SoC) while Figure 2.8 depicts its Vivado Block Design. Moreover, this SoC will be used to host the custom-designed cores needed to run the inference in hardware. The basic SoC is composed of the Zynq7 Processing System, the AXI interconnect, the Processor System Reset, and the AXI GPIOs. The Zynq7 Processing System manages the ARM processor, and it is configured as follows:

- **Zynq Block Design:** Default.
- **PS-PL Configuration:** Default.
- **Peripherals I/O Pins:** Default.
- **MIO Configuration:**
 - Bank 0 I/O Voltage: LVCMOS 3.3V.
 - Bank 1 I/O Voltage: LVCMOS 1.8V.
 - * Memory interfaces: Default.
 - * I/O Peripherals:

† SD0:

{ MIO 47, signal=cd, LVCMOS1.8V, speed=slow, pullup=disable, in.
MIO 40, signal=clk, LVCMOS1.8V, speed=fast, pullup=disable, inout.
MIO 41, signal=cmd, LVCMOS1.8V, speed=fast, pullup=disable, inout.
MIO 42, signal=data[0], LVCMOS1.8V, speed=fast, pullup=disable, inout.
MIO 43, signal=data[1], LVCMOS1.8V, speed=fast, pullup=disable, inout.
MIO 44, signal=data[2], LVCMOS1.8V, speed=fast, pullup=disable, inout.
MIO 45, signal=data[3], LVCMOS1.8V, speed=fast, pullup=disable, inout.

† UART1:

{ MIO 48, signal=tx, LVCMOS1.8V, speed=slow, pullup=enable, out.
MIO 49, signal=rx, LVCMOS1.8V, speed=slow, pullup=enable, in.

† GPIO/GPIO MIO:

{ MIO 07, signal=gpio[7], LVCMOS3.3V, speed=slow, pullup=disable, out.
MIO 50, signal=gpio[50], LVCMOS1.8V, speed=slow, pullup=enable, inout.
MIO 51, signal=gpio[51], LVCMOS1.8V, speed=slow, pullup=enable, inout.

- **Clock Configuration:** Default.
- **DDR Configuration:**
 - DDR Controller Configuration:
 - * Memory Type: DDR 3.
 - * Memory Part: MT41K256M16 RE-125.
 - Memory Part Configuration: Default.
- **SMC Timing Calculation:** Default.
- **Interrupts:** Default.

After setting up this configuration, click on “*Run Block Automation*” and Vivado will connect the Processor System Reset block. Besides, add the AXI interconnect and AXI GPIO blocks and click on “*Run Connection Automation*”. On the other hand, the input images and training parameters are 4D tensors stored on a microSD card as binary files. The program designed to create the binary files can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/microSD/M6. Also, algorithm 1 lists the function used to read the 4D tensors, which is located in https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/peripheral.c.

For computing the inference in hardware, custom cores were designed. These cores target the following layer operations: MultiplyByQuantizedMultiplier (mbqm), Convolution, MaxPooling and Dense or Fully Connected. In the following subsections, these cores will be presented.

Algorithm 1 Function to read 4D tensors from the microSD.

```
1: function ReadTensorSD(FileName, FileSize, TensorShape, tensor[x, y, z, w]){
2:   f_mount(&fatfs, Path, 0);
3:   f_open(&fil, SD_File, FA_READ);
4:   f_lseek(&fil, 0);
5:   f_read(&fil, (void*)DestinationAddress, FileSize, &NumBytesRead);
6:   ***
7:   for(int i = 0; i < TensorShape[0]; i ++){
8:     for(int j = 0; j < TensorShape[1]; j ++){
9:       for(int k = 0; k < TensorShape[2]; k ++){
10:        for(int l = 0; l < TensorShape[3]; l ++){
11:          tensor[i][j][k][l] =
12:            (DestinationAddress[0x83 + idx * 8] << 24) |
13:            (DestinationAddress[0x82 + idx * 8] << 16) |
14:            (DestinationAddress[0x81 + idx * 8] << 8) |
15:            (DestinationAddress[0x80 + idx * 8]); };
16:        }; }; };
17:   f_close(&fil); };
18: end function
```

3.2 Tflite Core

The tflite core was designed to evaluate the quantization functions involved in the inference. Algorithms 2, 3, and 4 show some of these functions. Besides, these functions were implemented in C++ and linked to Python by means of the ctypes library (see https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/02_M6_jaffe/01_M6_jaffe_tflite/mes_fon_CPP/src/monbib.cpp). For instance, the *SaturatingRoundingDoublingHighMul* function saturates the product between the input value (a) and the quantized_multiplier (b). The output value is bound to the int32_t maximum value. The exponent parameter is used by function *RoundingDivideByPOT* to round the saturated value. Lastly, *MultiplyByQuantizedMultiplier* calls the aforementioned function and passes the exponent derived from the shift quantization parameter.

Algorithm 2 Function *SaturatingRoundingDoublingHighMul* adapted from Tflite source code.

```
1: function SaturatingRoundingDoublingHighMul(std :: int32_t a, std :: int32_t b){
2:   bool overflow = a == b && a == std :: numeric_limits < std :: int32_t > :: min();
3:   std :: int64_t a_64(a); std :: int64_t b_64(b);
4:   std :: int64_t ab_64 = a_64 * b_64;
5:   std :: int32_t nudge = ab_64 >= 0 ? (1 << 30) : (1 - (1 << 30));
6:   std :: int32_t ab_x2_high32 = static_cast < std :: int32_t > ((ab_64 + nudge) / (1ll << 31));
7:   return overflow ? std :: numeric_limits < std :: int32_t > :: max() : ab_x2_high32; };
8: end function
```

In figure 2.9 the Tflite Core is shown. The core was designed to run the tflite stage of the inference and is described in VHDL. Its source files can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/ip_repo/tflite_mbqcm_1.0. Mainly, the core is composed of the 5 submodules described below:

Algorithm 3 Function `RoundingDivideByPOT` adapted from `TfLite`.

```

1: function RoundingDivideByPOT(std :: int32_t x, std :: int8_t exponent){
2:   assert(exponent >= 0);
3:   assert(exponent <= 31);
4:   const std :: int32_t mask = Dup((1ll << exponent) - 1);
5:   const std :: int32_t zero = Dup(0);
6:   const std :: int32_t one = Dup(1);
7:   const std :: int32_t remainder = BitAnd(x, mask);
8:   const std :: int32_t threshold = Add( ShiftRight(mask, 1),
9:                                       BitAnd(MaskIfLessThan(x, zero), one));
10:  return Add( ShiftRight(x, exponent),
11:            BitAnd( MaskIfGreaterThan(remainder, threshold), one ) ); };
12: end function

```

Algorithm 4 Function `MultiplyByQuantizedMultiplier` adapted from `TfLite`.

```

1: function MultiplyByQuantizedMultiplier(std :: int32_t x,
2:                                       std :: int32_t quantized_multiplier, int shift){
3:   std :: int8_t left_shift = shift > 0 ? shift : 0;
4:   std :: int8_t right_shift = shift > 0 ? 0 : -shift;
5:   return RoundingDivideByPOT(
6:     SaturatingRoundingDoublingHighMul(x * (1 << left_shift),
7:     quantized_multiplier), right_shift); };
8: end function

```

- **tfLite_core0**: For adding the *bias* to the input value (e.g. *cv_in*) and checking the presence of overflow.
- **tfLite_core1**: For multiplying the *quantized_multiplier* with the input value plus bias (i.e., signal *xls*). The operation is performed by 2 DSP48 because the result is 64 bits width. The *nudge* is also computed by this submodule.
- **tfLite_core2**: For adding the *ab_64* value to the *nudge* and storing to *ab_nudge*.
- **tfLite_core3**: For saturating the *ab_nudge* and binding it to the *int32_t* maximum value. The result is stored in the register *srdhm*.
- **tfLite_core4**: For rounding *srdhm* subject to the shift quantization parameter. The shift value is adapted by *MultiplyByQuantizedMultiplier*. The result is stored in *mbqm*.

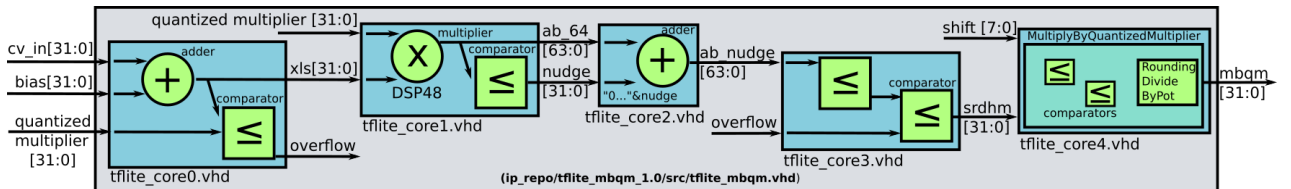


Figure 2.9: TfLite Core.

Figure 2.10 shows the output value = $0x\text{FFFF7C2}$ calculated with the input value = $0x\text{FFE9248}$, the bias = $0000004B$, the shift = FB , and the quantized_multiplier (M0) = $5C5D83CE$. This procedure reproduces the Tflite behavior on hardware and takes approximately $600\ \mu\text{s}$. *start* and *done* signals are employed to interact with the SoC.

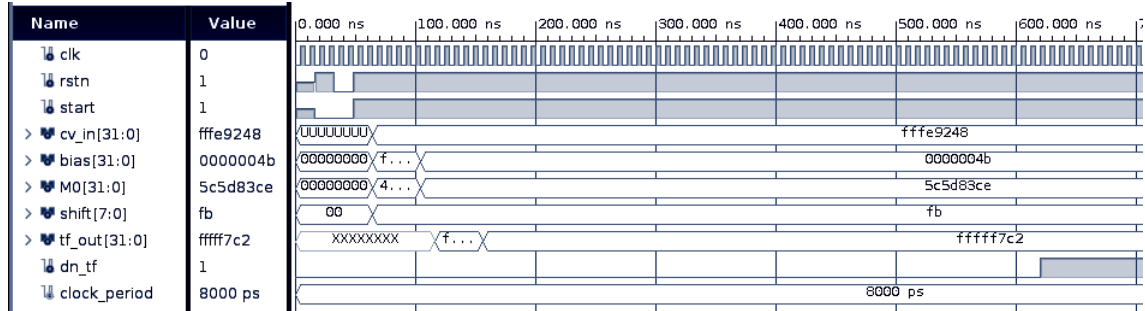


Figure 2.10: Tflite Core Simulation.

The core is connected to an AXI-Lite bus by means of a wrapper. The wrapper has registers for each core signal; hence, 7 registers were used. Each register address depends on the core BaseAddress assigned in the SoC. The algorithm 5 presents the function used to read and write the core registers from the ARM processor. The code for this function can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/accel.c. This function will be used by the custom cores designed for the Convolution and Dense operations.

Algorithm 5 Function to write and read the tflite core registers.

```

1: function tflite_mbqm(BaseAddress, cv_in, bias, M0, shift){
2:   TFLITE_MBQM_mWriteReg(BaseAddress,
3:     TFLITE_MBQM_S00_AXI_SLV_REG1_OFFSET, cv_in);
4:   TFLITE_MBQM_mWriteReg(BaseAddress,
5:     TFLITE_MBQM_S00_AXI_SLV_REG2_OFFSET, bias);
6:   TFLITE_MBQM_mWriteReg(BaseAddress,
7:     TFLITE_MBQM_S00_AXI_SLV_REG3_OFFSET, M0);
8:   TFLITE_MBQM_mWriteReg(BaseAddress,
9:     TFLITE_MBQM_S00_AXI_SLV_REG4_OFFSET, shift);
10:  ***
11:  while( TFLITE_MBQM_mReadReg(BaseAddress, /* done signal */
12:    TFLITE_MBQM_S00_AXI_SLV_REG6_OFFSET) == 0x00000000 ){};
13:  tf_out = TFLITE_MBQM_mReadReg(BaseAddress,
14:    TFLITE_MBQM_S00_AXI_SLV_REG5_OFFSET);
15:  return tf_out; };
16: end function

```

3.3 Conv Core

The function for computing the Tflite convolution operation in Numpy is shown in the algorithm 6, while the complete code can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/02_M6_jaffe/01_M6_jaffe_tflite/mes_fonctions.py Observe that *entree* represents a tensor with the input data, and *fil* is the weight tensor for each filter. In addition, the output values are stored in the tensor *cv.tab*. And, for quantization and inference other variables (e.g., *shift*, *M0*, *scale*, *offset_ent*, *offset_sor*) are used. And the *QuantizeMultiplier* function was directly mapped from the Tflite source code. The first clamp operation (see line 12) represents a ReLU from 0 to 255. In the second clamp (line 14), the *min_val=-128* and *max_val=127* correspond to the *int8* range.

Algorithm 6 Tflite Convolution implemented in Numpy.

```

1: function conv_k5_npT(entree, filtros, biases, M, scale, offset_ent, offset_sor) :
2:   for f in range(0, #filters) :
3:     shift, M0 = QuantizeMultiplier(M[f]/scale);
4:     for i in range(0, #rows - 4) :
5:       for j in range(0, #cols - 4) :
6:         for k in range(0, #cc - 4) :
7:           ent = entree[0, 0 + i : 5 + i, 0 + j : 5 + j, k];
8:           ent = ent + offset_ent;
9:           cv.tab[0, i, j, f] = cv.tab[0, i, j, f] + np.tensordot(ent, filtros[f, :, :, k];
10:          cv.tab[0, i, j, f] = cv.tab[0, i, j, f] + biases[f];
11:          cv.tab[0, i, j, f] = QuantizeMultiplier(cv.tab[0, i, j, f], M0, shift);
12:          cv.tab[0, i, j, f] = min(max(cv.tab[0, i, j, f], 0), 255);
13:          cv.tab[0, i, j, f] = cv.tab[0, i, j, f] + offset_sor;
14:          cv.tab[0, i, j, f] = min(max(cv.tab[0, i, j, f], min_val), max_val);
15:   return cv.tab;
16: end function

```

In figure 2.11 the Conv Core is presented. This core is capable of computing a convolutional window of 25×25 and is described in VHDL. Its source code can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/ip_repo/conv_1.0. The core is divided into 4 submodules:

- **conv_core0:** For adding the *offset_ent* to the input values (*x*). *xo* is the name given to the output signals.
- **conv_core1:** For multiplying the weights (*w*) with the *xo* values using DSP48. The output signals are labeled as *xow*.
- **conv_core2:** For adding all the *xow* values into one *xow* signal.
- **conv_core3:** For adding the previous value *cv_in* to the present value *xow*. This result is saved in the register *cv_out*.

To validate the core behavior, several operations were simulated. For instance, the simulation in figure 2.12 shows that each convolution made takes approximately 500 ns.

The core is connected to an AXI-Lite bus by means of a wrapper. The wrapper has registers for each core signal; hence, 55 registers were used. Likewise, each register address depends on the core's BaseAddress assigned

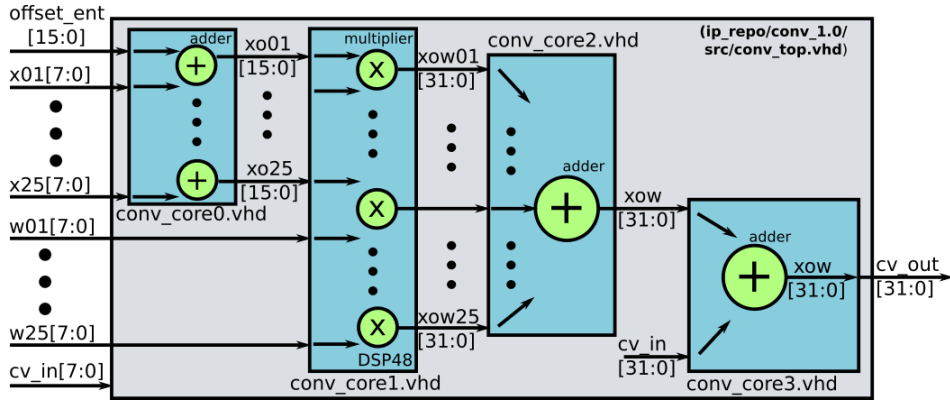


Figure 2.11: Conv Core.

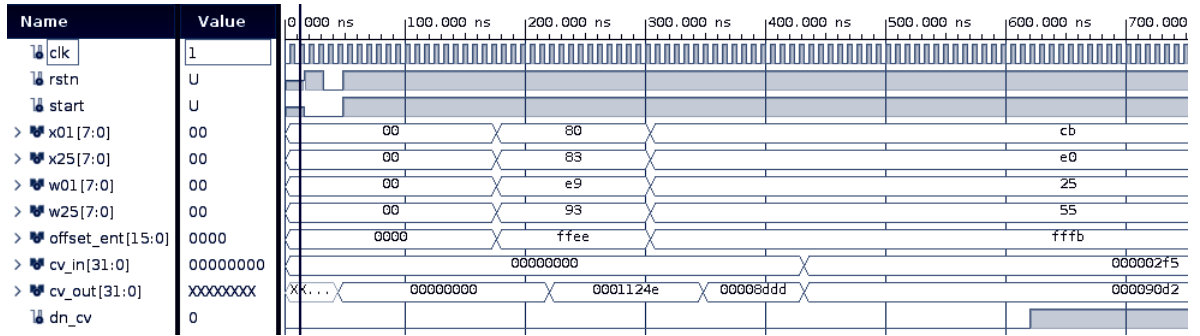


Figure 2.12: Conv Core Simulation.

in the SoC. The algorithm 7 describes the function *cv_k5_core* used to read and write the core registers from the ARM processor.

Algorithm 7 Function to write and read the Conv core registers.

```

1: function cv_k5_core(BaseAddress, x01, ..., x25, w01, ..., w25, offset_ent, cv_in){
2:   CONV_mWriteReg(BaseAddress, CONV_S00_AXI_SLV_REG1_OFFSET, x01);
3:   * **
4:   CONV_mWriteReg(BaseAddress, CONV_S00_AXI_SLV_REG26_OFFSET, w01);
5:   * **
6:   CONV_mWriteReg(BaseAddress, CONV_S00_AXI_SLV_REG51_OFFSET, offset_ent);
7:   CONV_mWriteReg(BaseAddress, CONV_S00_AXI_SLV_REG52_OFFSET, cv_in);
8:   * **
9:   cv_out = CONV_mReadReg(BaseAddress, CONV_S00_AXI_SLV_REG53_OFFSET);
10:  return cv_out; }
11: end function

```

The algorithm 8 shows the function *conv_k5* executed by the ARM processor. *ent* is the input tensor with dimensions *x,y,z,w,fil* is the filters tensor, *par* is the parameters tensor, and *cnv* the output tensor.

The function is constantly employing the *conv* and the *tf-mbqm* cores to compute values. This is attained by means of the functions *cv_k5_core* and *tflite_mbqm*. As the *cnv* tensor is stored in RAM, it is available for the next layer. These functions can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/accel.c.

Algorithm 8 Function to compute a convolutional layer

```

1: function conv_k5(EntShape, ent[x,y,z,w], FilShape, fil[x,y,z,w],
2:                 ParShape, par[x,y,z,w], CnvShape, cnv[x,y,z,w]){
3:   for(f = 0; f < FilShape[0]; f ++){
4:     get: shift, M0, bias
5:     for(i = 0; i < EntShape[1] - 4; i ++){
6:       for(j = 0; j < EntShape[2] - 4; j ++){
7:         for(k = 0; k < FilShape[3]; k ++){
8:           cnv[0][i][j][f] = cv_k5_core(BaseAddress0, ent[0,0 + i,0 + j,k], ..., fil[f,0,0,k], ...,
9:                                     offset_ent, cnv[0][i][j][f]) };
10:          cnv[0][i][j][f] = tflite_mbqm(BaseAddress1, cnv[0][i][j][k], bias, M0, shift);
11:          cnv[0][i][j][f] = min( max(cnv[0][i][j][f], 0), 255);
12:          cnv[0][i][j][f] = cnv[0][i][j][f] + offset_sor;
13:          cnv[0][i][j][f] = min( max(cnv[0][i][j][f], -128), 127); };
14:        }; };
15:     return cnv; };
16: end function

```

3.4 MaxPooling Core

The maxpool core was designed to implement the *maxpool_npT* function for a kernel size of 2×2 . The function takes four values and returns the maximum one. Then it repeats that process across the entire input data. Usually, the input data could be the feature maps obtained in the convolutional layer. The function is shown in algorithm 9 and can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/02_M6_jaffe/01_M6_jaffe_tflite/mes_fonctions.py.

Algorithm 9 Tflite MaxPooling implemented in Numpy.

```

1: function maxpool_npT(entree) :
2:   ***
3:   for i in range(0, #rows/2) :
4:     for j in range(0, #cols/2) :
5:       for k in range(0, #cc/2) :
6:         mp_tab[0,i,j,k] = np.max( entree[0,2 * i : 2 * i + 2,2 * j : 2 * j + 2,k] );
7:       return mp_tab;
8:   end function

```

The core design is shown in figure 2.13. It takes four values at the same time, compares them, and outputs their maximum one. The core is written in VHDL, and its source files can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/ip_repo/mpool_1.0/src/maxpool.vhd

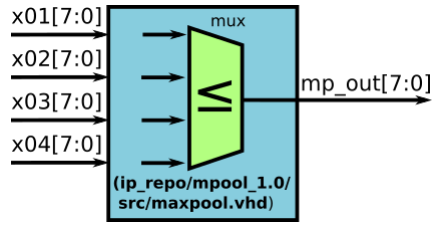


Figure 2.13: MaxPooling Core.

The core is connected to an AXI-Lite bus by means of a wrapper. The wrapper has registers for each core signal; hence, 7 registers were used. Each register address depends on the core BaseAddress assigned in the SoC. Algorithm 10 describes the *mp_22_core* function used to read and write the core's registers from the ARM processor. This function takes four values as input parameters and returns the maximum. In addition, note that the BaseAddress points to the core.

Algorithm 10 Function to write and read the mp_22 core registers.

```

1: function mp_22_core(BaseAddress, x01, x02, x03, x04){
2:   MPOOL_mWriteReg(BaseAddress, MPOOL_S00_AXI_SLV_REG1_OFFSET, x01);
3:   MPOOL_mWriteReg(BaseAddress, MPOOL_S00_AXI_SLV_REG2_OFFSET, x02);
4:   MPOOL_mWriteReg(BaseAddress, MPOOL_S00_AXI_SLV_REG3_OFFSET, x03);
5:   MPOOL_mWriteReg(BaseAddress, MPOOL_S00_AXI_SLV_REG4_OFFSET, x04);
6:   * * *
7:   while( MPOOL_mReadReg(BaseAddress, /* done signal */
8:                       MPOOL_S00_AXI_SLV_REG6_OFFSET) == 0x00000000 ){};
9:   res = MPOOL_mReadReg(BaseAddress,
10:                      MPOOL_S00_AXI_SLV_REG5_OFFSET);
11:   return res; };
12: end function

```

Moreover, the function *mp_22_core* is called by the *maxp_22* function presented in the algorithm 11, which goes through the input data and creates the tensor from the MaxPooling layer. *cnv* is the input tensor coming from the convolutional layer (i.e., thus the feature maps), and its dimensions are represented by x,y,z,w . *mxp* is the output tensor for the MaxPooling layers. The tensor is stored in RAM, so it is available for the next layer. The code for these functions can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/accel.c.

3.5 Dense Core

The function for computing the Tflite dense operation in Numpy is depicted in the algorithm 12 and its entire code can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/00_sw/02_M6_jaffe/01_M6_jaffe_tflite/mes_fonctions.py. Observe that *entree* is a vector with the input data, *fil* is the weights matrix, and the output values are stored in the vector *fc_vec*. Other variables (i.e., *shift*, *M0*, *scale*, *offset_ent*, *offset_sor*) are used for quantization and inference. And the *QuantizeMultiplier* function was directly mapped from the Tflite source code.

Algorithm 11 Function to compute the maxpooling layer.

```

1: function maxp_22(CnvShape, cnv[x, y, z, w], MxpShape, mxp[x, y, z, w]){
2:   for(i = 0; i < MxpShape[1]; i ++){
3:     for(j = 0; j < MxpShape[2]; j ++){
4:       for(k = 0; k < MxpShape[3]; k ++){
5:         mxp[0][i][j][k] =
6:           mp_22_core(BaseAddress,
7:                     cnv[0, 0 + i * 2, 0 + j * 2, k], cnv[0, 0 + i * 2, 1 + j * 2, k],
8:                     cnv[0, 1 + i * 2, 0 + j * 2, k], cnv[0, 1 + i * 2, 1 + j * 2, k]);
9:       }; }; };
10: end function

```

In the clamp at line 9, the *min_val*=-128 and *max_val*=127 correspond to the *int8* range.

Algorithm 12 Tflite Dense implemented in Numpy.

```

1: function dense_npT(entree, params, biases, M, scale, offset_ent, offset_sor):
2:   fc_vec = []; # list of classification.
3:   for cls in range(0, #classes):
4:     shift, M0 = QuantizeMultiplier(M[0]/scale);
5:     ent = ent + offset_ent;
6:     Wx_b = np.tensordot([ent[0]], [params[cls]]) + biases[cls];
7:     Wx_b = QuantizeMultiplier(Wx_b, M0, shift);
8:     Wx_b = Wx_b + offset_sor;
9:     Wx_b = min(max(Wx_b, min_val), max_val);
10:    fc_vec.append(Wx_b);
11:  return fc_vec;
12: end function

```

In figure 2.14 the Dense Core is presented. It's capable of computing the main operation of a dense or full connected layer and is described in VHDL. Despite the fact that typical dense layers involved a large number of factors, the core was designed to operate 64×64 elements at once. Its source code can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/ip_repo/dense_1.0. The core is divided into two submodules:

- **dense_core0**: For adding the *offset_ent* to the input values (*x*). *xo* is the name given to the output signals.
- **dense_core1**: For multiplying the weights (*w*) with the *xo* values using DSP48. The output signals are labeled as *xow*, and are added into the top module. Its result will be saved in *ds_out*.

The core is connected to an AXI-Lite bus by means of a wrapper. The wrapper has registers for each core signal; hence, 133 registers were used. Each register address depends on the core BaseAddress assigned in the SoC. The algorithm 13 describes the function *ds_k64_core* employed to read and write the core's registers from the ARM processor.

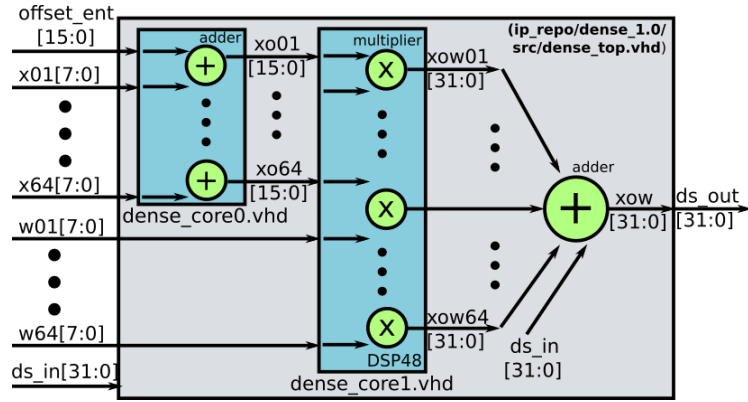


Figure 2.14: Dense Core.

Algorithm 13 Function to write and read the Dense core registers .

```

1: function ds_k64_core(BaseAddress, x01, ..., x64, w01, ..., w64, offset_ent, ds_in){
2:   DENSE_mWriteReg(BaseAddress, DENSE_S00_AXI_SLV_REG1_OFFSET, x01);
3:   ***
4:   DENSE_mWriteReg(BaseAddress, DENSE_S00_AXI_SLV_REG65_OFFSET, w01);
5:   ***
6:   DENSE_mWriteReg(BaseAddress, DENSE_S00_AXI_SLV_REG129_OFFSET, offset_ent);

7:   DENSE_mWriteReg(BaseAddress, DENSE_S00_AXI_SLV_REG130_OFFSET, ds_in);
8:   ***
9:   ds_out = DENSE_mReadReg(BaseAddress, DENSE_S00_AXI_SLV_REG131_OFFSET);

10:  return ds_out; };
11: end function

```

The algorithm 14 represents the function *dense* used to compute the dense layer. *ent* is the input vector with dimension x . *fil* is the filters matrix, *par* the parameters matrix, and *dns* the output vector. The function is constantly employing the *dense* and the *tf_mbqm* cores to compute values by means of the functions *ds_k5_core* and *tfite_mbqm*. These functions can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/accel.c.

3.6 Additional functions

Other functions needed to run the inference in hardware are the *padding* and *flatten*. The *padding* function is shown in the algorithm 15. It adds elements with zero value to the tensor to maintain the same size between the layer's input and output tensors. As the zero value is affected by the quantization process, the value used is the *zero_point* parameter.

Although this process could be memory-intensive, it can be improved by adding execution conditions through the layers. However, this will increase the core's complexity and could lead to more hardware resources being needed. As the hardware platform employed has enough memory (i.e., 1 GB DDR3), this implementation is affordable.

Algorithm 14 Function to compute a dense layer

```
1: function dense(EntShape, ent[x], FilShape, fil[x, y],
2:               ParShape, par[x, y], DnsShape, dns[x]){
3:   for(f = 0; f < FilShape[0]; f ++){
4:     get: shift, M0, bias
5:     for(i = 0; i < EntShape[0]/64; i ++){
6:       dns[f] = ds_k64_core(BaseAddress0,
7:                           ent[0 + 64 * i], ..., ent[63 + 64 * i],
8:                           fil[f][0 + 64 * i], ..., fil[f][63 + 64 * i]
9:                           offset_ent, dns[f]); };
10:    dns[f] = tflite_mbqm(BaseAddress1, dns[f], bias, M0, shift);
11:    dns[f] = dns[f] + offset_sor;
12:    dns[f] = min( max(dns[f], -128), 127); };
13:   return dns; };
14: end function
```

Algorithm 15 Function to compute the padding layer.

```
1: function padding(EntShape, ent[x, y, z, w], PadShape,
2:               pad[x, y, z, w], zero_point){
3:   for(f = 0; f < EntShape[0]; f ++){
4:     for(i = 0; i < EntShape[1]; i ++){
5:       for(j = 0; j < EntShape[2]; j ++){
6:         for(k = 0; k < EntShape[3]; k ++){
7:           if( ( (i >= 0) && (i < 2)) || ((i >= EntShape[1] + 2) && (i < EntShape[1] + 4)) ) ||
8:             ( (j >= 0) && (j < 2)) || ((j >= EntShape[2] + 2) && (i < EntShape[2] + 4)) ) ){
9:             pad[f, i, j, k] = zero_point; };
10:          else{
11:            pad[f, i, j, k] = ent[f, i - 2, j - 2, k]; };
12:          }; }; }; }; };
13: end function
```

The *flatten* function described in algorithm 16 takes a tensor as input and creates a 1D array. Its dimensions will be subject to the number of feature maps, their size, and the number of classes. The output vector will be stored in the SoC RAM and will be the input for a Dense layer.

Algorithm 16 Function to compute the flatten layer.

```
1: function flatten(EntShape, ent[x, y, z, w], FltShape, flt[x]){
2:   int idx = 0;
3:   for(i = 0; i < EntShape[1]; i ++){
4:     for(j = 0; j < EntShape[2]; j ++){
5:       for(k = 0; k < EntShape[3]; k ++){
6:         flt[idx] = ent[0, i, j, k];
7:         idx = idx + 1; };
8:       }; }; };
9: end function
```

4 Putting all together: the Hardware Neural Network Accelerator

4.1 FER SoC design

Running the inference of a CNN model in an FPGA requires a SoC with specific cores. In this section, the base SoC and the cores presented before will be integrated. This integration will be known as the FER SoC. In addition, its block diagram is shown in figure 2.15, along with its Vivado block design in figure 2.16.

Here, the specific cores reproduce the operations of convolution, maxpooling, dense or fullconnected, and MultiplyByQuantizedMultiplier following the TensorFlow Tflite guidelines. Furthermore, the cores are connected through an AXI interconnect interface to the ARM processor. Hence, each core is controlled by reading and writing their registers using their base addresses.

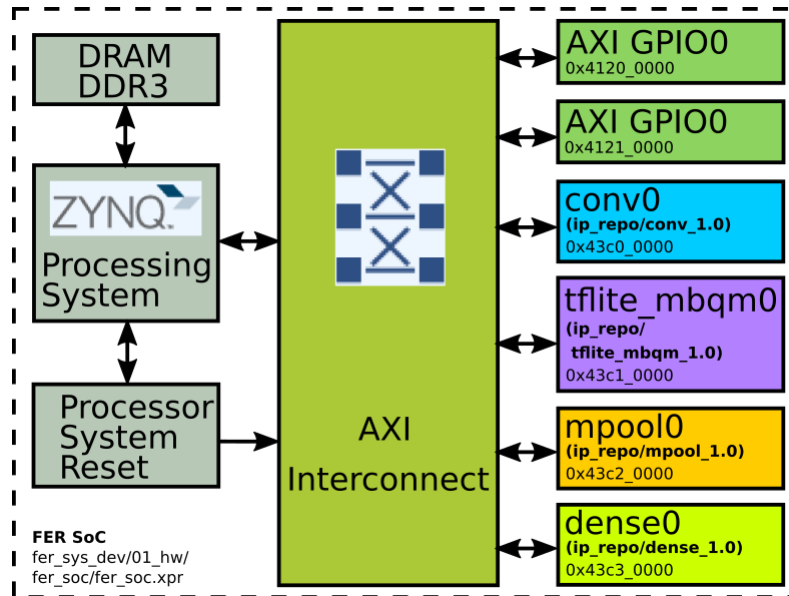


Figure 2.15: FER SoC.

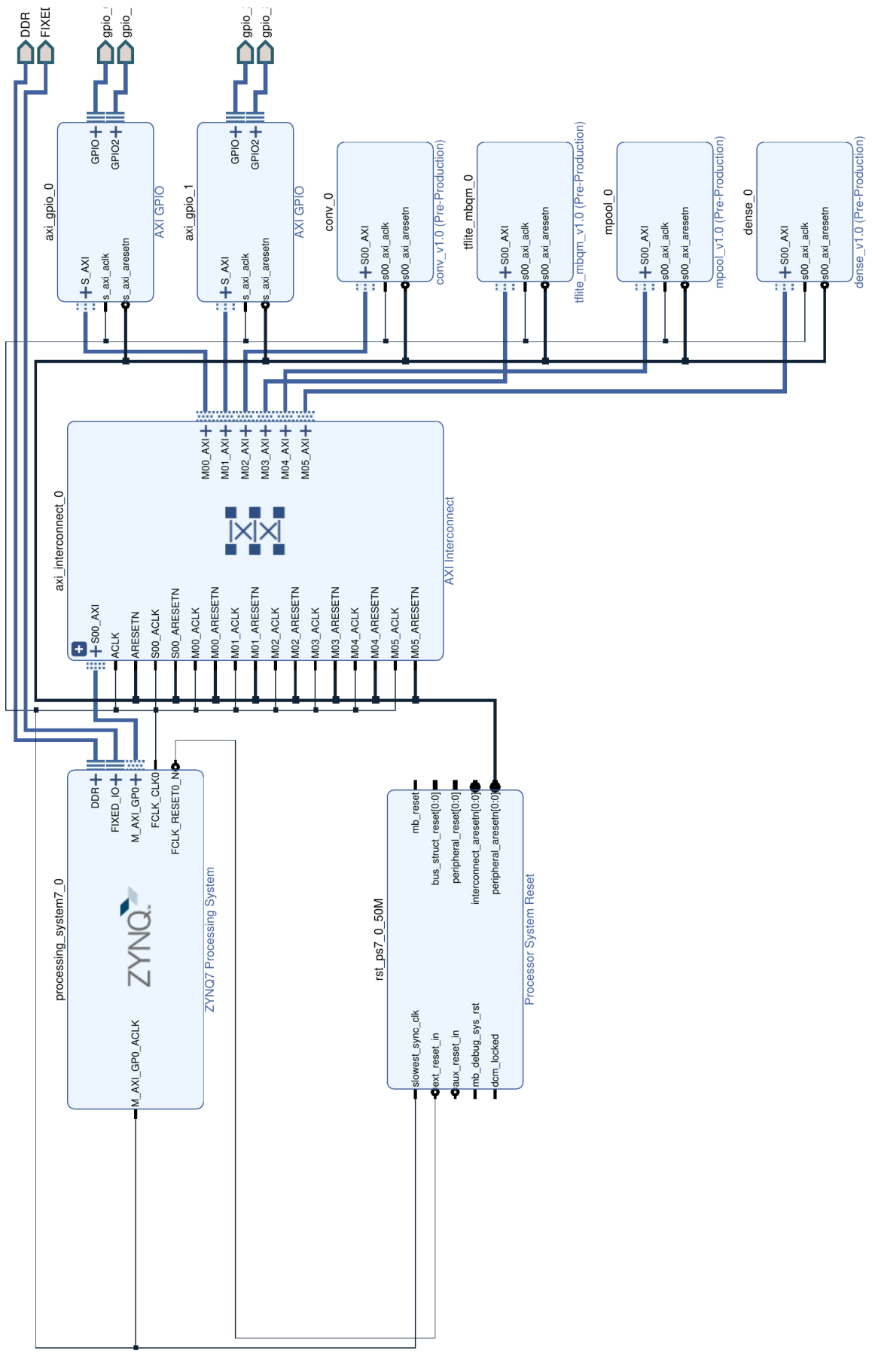


Figure 2.16: Vivado Block Design for the FER SoC.

In figure 2.17 the base addresses employed for each core are shown. Besides, the Vivado project with the FER SoC design can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/fer_soc.

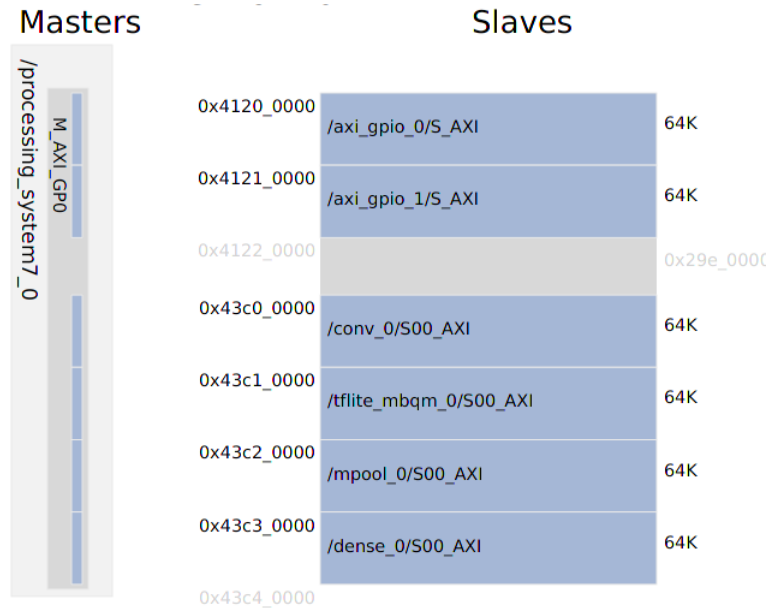


Figure 2.17: Memory map for the FER SoC.

Figure 2.18 shows the FER SoC placed and routed into the FPGA. The number of resources employed is summarized in Table 2.5. Despite the fact that parallelism could be improved by adding cores' instances, it was observed that large cores make the routing unfeasible because there isn't enough space in the FPGA used.

Resource	Available	Utilization	Utilization %
LUT	53200	6373	11.98
LUTRAM	17400	71	0.41
FF	106400	12470	11.72
DSP	220	93	42.27
IO	125	18	14.40

Table 2.5: FER SoC Utilization.

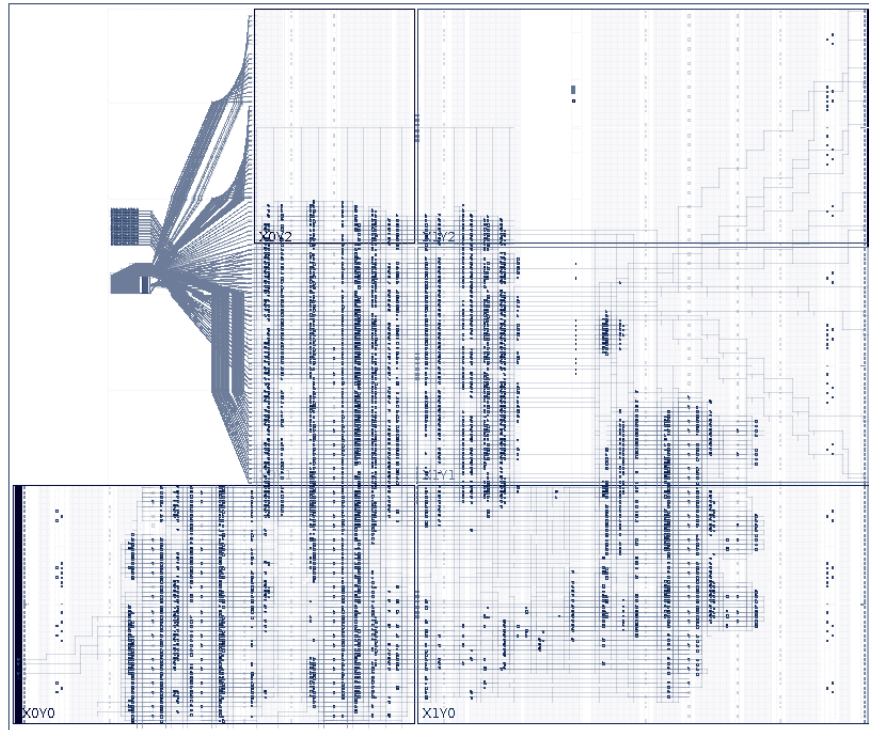


Figure 2.18: FER SoC place and route.

Furthermore, Figure 2.19 shows the Vivado estimation for the FER SoC power consumption. Here, the Processing System (i.e., the ARM processor) had the higher consumption: 1.53 W, and the total estimated power is less than 1.7 W.

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 1.695 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 44.6°C
 Thermal Margin: 40.4°C (3.4 W)
 Effective θ_{JA} : 11.5°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Low

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

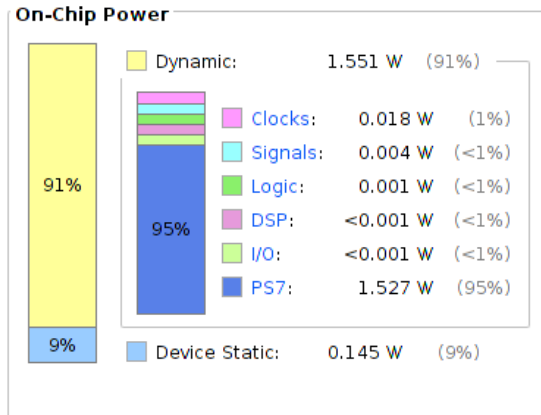


Figure 2.19: FER SoC power consumption.

4.2 FER SoC testing

Once the FER SoC has been implemented, the next step is to design the software routines needed to execute the CNN model. This interaction between the hardware and software will be known as the accelerator.

Furthermore, the Zynq7 ARM processor will be the accelerator's master. Thus, it will run the inference function and keep track of the execution process. This architecture can take advantage of the hardware processor latency and the FPGA throughput.

Algorithm 17 Function to execute the M6 inference.

```
1: function M6_inference(InShape, InTensor[x, y, z, w]){
2:   /** layer1 **/
3:   padding(InShape, InTensor, Pad1Shape, Pad1Tensor, zero_point);
4:   conv_k5(Pad1Shape, Pad1Tensor, Fil1Shape, Fil1Tensor,
5:           Par1Shape, Par1Matrix, Cnv1Shape, Cnv1Tensor);
6:   maxp_22(Cnv1Shape, Cnv1Tensor, Mxp1Shape, Mxp1Tensor);
7:   /** layer2 **/
8:   padding(Mxp1Shape, Mxp1Tensor, Pad2Shape, Pad2Tensor, zero_point);
9:   conv_k5(Pad2Shape, Pad2Tensor, Fil2Shape, Fil2Tensor,
10:          Par2Shape, Par2Matrix, Cnv2Shape, Cnv2Tensor);
11:  maxp_22(Cnv2Shape, Cnv2Tensor, Mxp2Shape, Mxp2Tensor);
12:  /** layer3 **/
13:  padding(Mxp2Shape, Mxp2Tensor, Pad3Shape, Pad3Tensor, zero_point);
14:  conv_k5(Pad3Shape, Pad3Tensor, Fil3Shape, Fil3Tensor,
15:         Par3Shape, Par3Matrix, Cnv3Shape, Cnv3Tensor);
16:  maxp_22(Cnv3Shape, Cnv3Tensor, Mxp3Shape, Mxp3Tensor);
17:  /** layer4 **/
18:  flatten(Mxp3Shape, Mxp3Tensor, FltShape, FltVector);
19:  /** layer5 **/
20:  dense(FltShape, FltVector, Fil5Shape, Fil5Matrix,
21:        Par5Shape, Par5Matrix, Dns1Shape, Dns1Vector);
22:  return 0; }
23: end function
```

In algorithm 17, the function *M6_inference* reproduces the architecture of model M6. Its complete definition can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/M6FER.c. The function employs the JAFFE LBP image and layer parameters, including the quantization ones. These tensors are read from the SD card and stored in the SoC RAM. Then, layers are computed by using the functions presented in the former section. These results will also be stored in the SoC RAM, so they can be used by the following layers. The inference ends with a vector containing the integer value assigned to each class. The greatest value will be the classification result.

Figure 2.20 presents how the accelerator proposed can be used to execute the model M6 inference. Although the accelerator is aimed at model M6, it is suitable for CNN models with similar characteristics. Also, the SoC firmware is compiled through a Vitis project located at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/fer_soc/firmware.

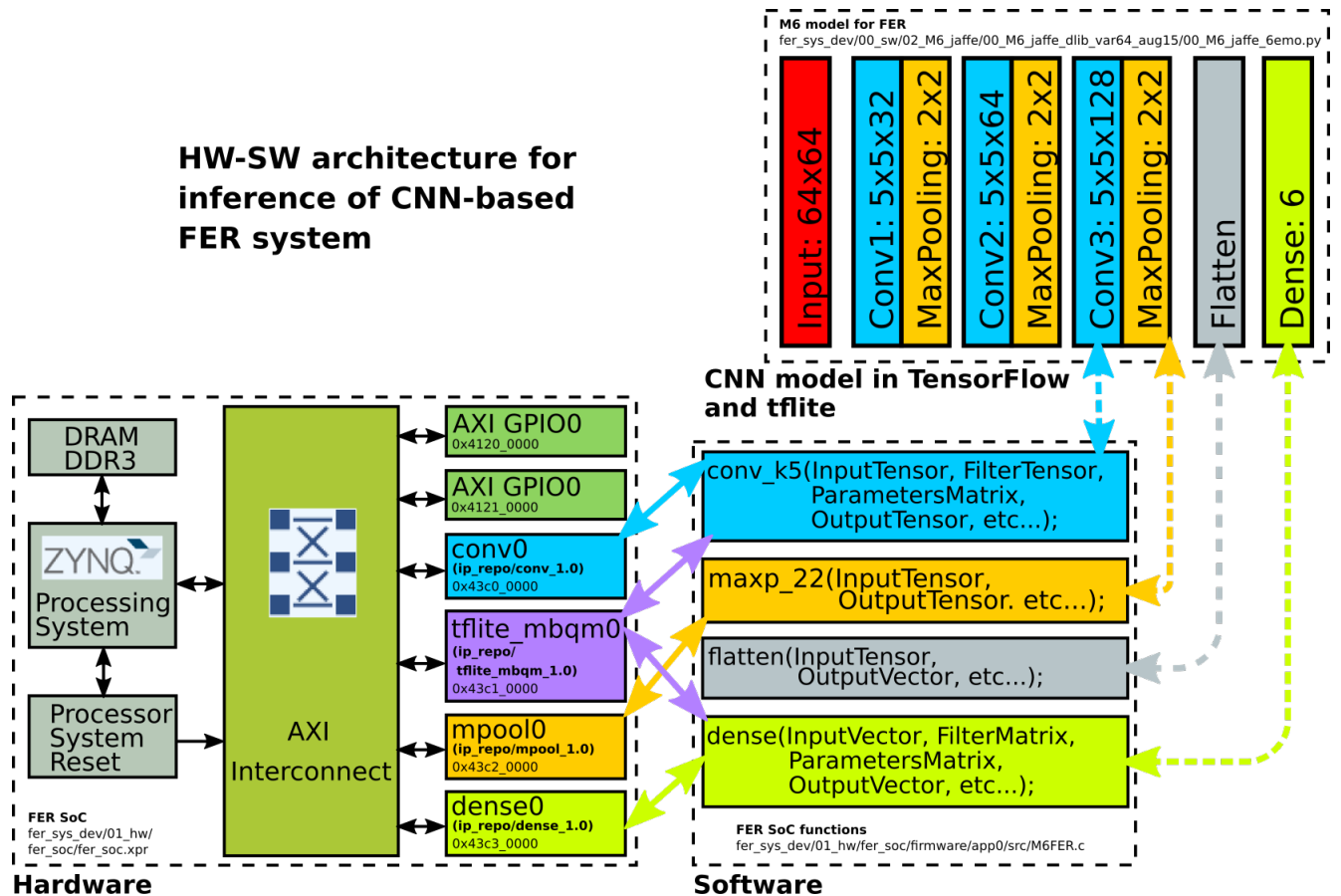


Figure 2.20: Running the inference of a CNN-based FER system in the accelerator.

Lastly, the accelerator results were compared to the values obtained by a laptop running the M6's inference with Numpy (see table 2.6). The AMD Ryzen7 4800H processor (16 threads up to 4.2 GHz), 16 GB of RAM, and a GPU Nvidia GeForce GTX 1660 Ti are among the laptop's specifications. The OS installed is Ubuntu 20.04.2 LTS, kernel version 5.13.0-37-generic, with TensorFlow 2.6.0. Its execution time was 82s. On the other hand, the FER SoC employs a XC7Z020-1CLG400C FPGA. Although this Zynq7 chip includes an 2 cores ARM processor running at 660 MHz, only one core is used. In addition, the Zybo-Z7 board provides a 1 GB DDR3 RAM. M6's inference took 99 s on the accelerator.

Besides, values from the two platforms were the same because of the integer representation used. Moreover, a demonstration of the M6's inference running in the accelerator and the laptop can be found at <https://www.youtube.com/watch?v=WQdaqADqCEE>. Also, a screenshot of this test has been added to Figure C.4.

Furthermore, it is worth mentioning that although the FER SoC is 17 s slower than the laptop, the FER SoC needs fewer resources and power. Besides, the FER SoC run on a cheaper hardware platform.

Platform Name	Hardware specs	Software	Execution time
Legion 5	Processor: AMD Ryzen 7 4800H RAM: 16 GB GPU: Nvidia GeForce GTX 1660 Ti.	OS: Ubuntu 20.04.2 LTS kernel: 5.13.0-37-generic library: Numpy.	82 s
Zybo-Z7	FPGA: Xilinx Zynq XC7Z020. Processor: ARM A9 core at 660 MHz RAM: 1 GB DDR3.	Baremetal Firmware: M6FER.c Vitis project	99 s

Table 2.6: M6 inference performance comparison.

5 Discussion

The inference of neural networks requires a large amount of memory and compute capability. Usually, large GPUs are employed, but they are costly and have high power consumption. Therefore, they could be unfeasible for some portable systems. On the other hand, specific SoCs implemented in FPGAs could consume less power and support pipelining. For instance, the FER SoC proposed is capable of computing up to 64 operations at the same time with less than 1.7 W.

Furthermore, parallelism could be improved by employing several cores' instances, or by using all the cores available in the ARM processor. Despite this, as mentioned in subsection 4 implementing the design in the FPGA needs space to place and route the paths. Thus, it's important to distribute the computational load between the processing system and the programmable logic.

6 Conclusion

This chapter proposes a hardware-software architecture (i.e., the accelerator) capable of executing the inference of a CNN model. This architecture employs the tflite quantization scheme. Achieving acceptable accuracy with integer parameters instead of decimal values.

Besides, IP cores suitable for computing pertinent operations for the M6's inference were designed. These cores make use of optimized hardware resources such as DPS48 and BRAMs, and they are integrated into the FER SoC. The last could be improved by adding more IP core instances or increasing the number of elements per operation. However, this customization will be limited by the FPGA's resources and the space available for the place and route process.

Furthermore, the accelerator's results and performance were compared to a laptop equipped with an AMD Ryzen7, Nvidia GPU and up to 16 GB RAM running Numpy in Ubuntu 20.04.

Although the accelerator was the slowest, the resources it used and its cost were the lowest. Therefore, the accelerator could be a feasible and attractive alternative when the trade-off between the execution time, resources and costs would be flexible.

Resiliency: A Framework for Training a CNN with a custom Hardware-Software (HW-SW) Architecture.

Although embedded systems are widely employed in several applications, their use in Machine Learning (ML) scenarios is commonly bounded to the inference stage. Besides, the training phase is avoided due to its implicit complexity and the number of logic resources required. However, training ML algorithms locally could be relevant in scenarios where new data samples must be considered at run-time or where previously trained models will be retrained, as in the Transfer Learning technique [22].

On the other hand, works found in the literature aimed at accelerating CNNs on hardware do not provide enough information to reproduce their results. The aforementioned evidences a gap in frameworks for training CNNs in custom hardware-software architectures. This work aims to reduce this gap with Resiliency: a framework that provides the design resources needed to train CNNs on embedded systems.

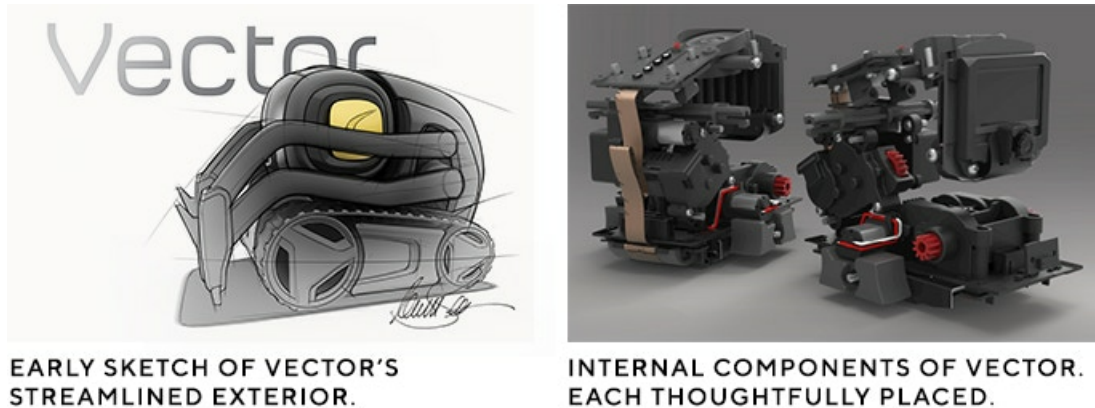
The chapter has the following structure: In section 1 a brief review of some custom hw-sw architectures that are commercially available are presented. Those devices cover virtual assistants and accelerators, among others. Then, section 2 presents the process of designing a hw-sw architecture for running a custom Pynq image. This image will provide the Python environment needed for the ML frameworks. Here, the popular frameworks TensorFlow and PyTorch are tested on the Zybo-Z7 board running the custom Pynq 2.7 image. These tests include simple CNN models trained with the MNIST and CIFAR10 data sets and the M6 model proposed for FER applications and trained with JAFFE. Performance metrics like accuracy, loss, and execution time are reported in section 3. Furthermore, section 4 describes the hw-sw architectures involved in Resiliency and how they can be used for inference and training. The framework also points out aspects that could improve the overall performance.

Moreover, the reader will find links to the source files, development projects, and test examples, which would make this work reproducible. Lastly, the discussion and conclusions can be found in sections 5 and 6.

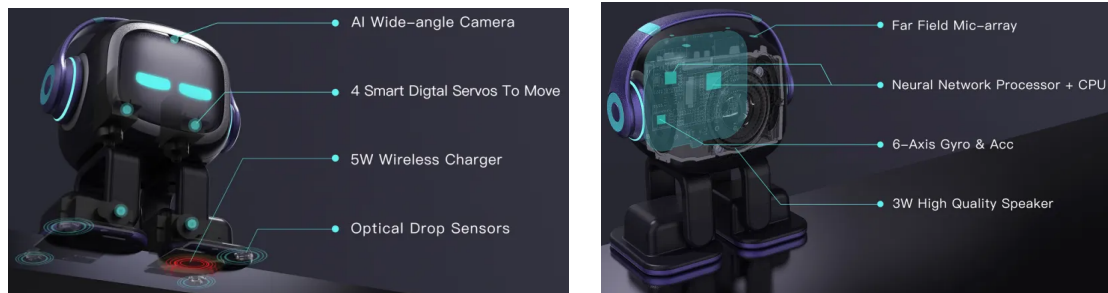
1 Custom hardware-software architectures aimed to ML algorithms

The training of CNNs usually involves loading into memory mini batches of data, layer parameters (e.g., weights, biases, etc.), and partial results. Besides, operations for computing the optimization stage and layers' outputs frequently demand complex hardware.

On the other hand, several ML applications, like virtual assistants and smart appliances, employ embedded systems that connect to online servers (e.g., Amazon Alexa [39], Google Assistant [40]). In addition, AI desktop pets like Vector [41], [42] and EMO [43] use these servers with traditional hardware or aim to include specific IP cores capable of executing the inference's algorithms locally. The cost of these platforms is around USD \$350 for Vector and USD \$279 for EMO, and some examples are shown in figure 3.1.



(a) Vector Robot is based on a Qualcomm 200 Processor. Taken from [41].

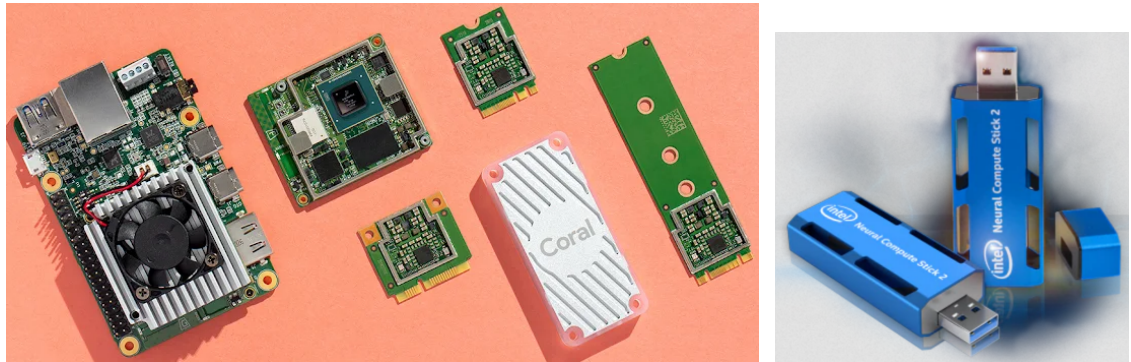


(b) EMO Robot uses a CPU with a NN processor. Taken from [43].

Figure 3.1: Examples of AI desktop pets: Vector by DDL and EMO by Living AI.

Moreover, specific hardware devices targeting Neural Networks' inference acceleration through handling tensor operations have been released. For instance, Figure 3.2 shows the Google Coral devices [44] costing up to USD \$130, and the Intel Neural Compute Stick 2 (NCS2) [45] costing around USD \$150. Despite their capabilities, they are constrained by data transfer bandwidth (e.g., USB 3.0), ML frameworks support, and the dependency on most of them of at least a laptop controlling the execution flow.

Furthermore, applications will eventually require model parameters that take data acquired at run time; thus, techniques that use pre-trained models (e.g., MobileNetV2) to retrain custom layers are appealing [22]. Nevertheless, from an embedded systems perspective, it is needed to have an affordable hardware-software



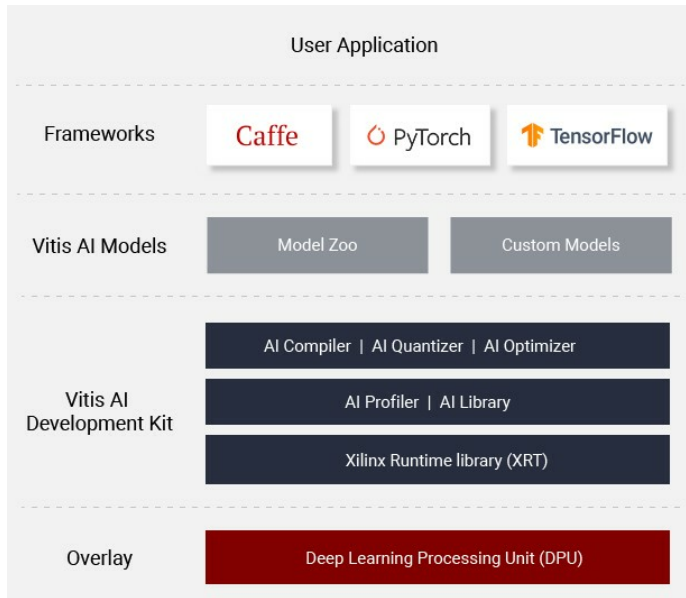
(a) Coral devices by Google. Taken from [44].

(b) Intel NCS2. Taken from [45]

Figure 3.2: Hardware devices for acceleration.

architecture capable of executing the training algorithm.

To target this issue, development environments such as Vitis AI [46] had been proposed. This environment is presented in figure 3.3a. The lowest level uses an overlay with a Deep Learning Processing Unit (DPU). The following level provides the Xilinx Runtime Library (XRT) and the AI compiler, Quantizer, and Optimizer. These levels provide support for ML frameworks like Caffe, PyTorch, and TensorFlow. However, the Xilinx hardware platforms supported are high-performance edge devices like the Alveo cards [47] shown in Figure 3.3b. Unfavorably, these devices are usually expensive and use large amounts of power (e.g., 75 W). In addition, some Xilinx IP cores require a paid license to generate the configuration files (e.g., bitstreams, firmware, etc.).



(a) Vitis AI environment. Taken from [46].



Active Option

(b) Alveo U200. Taken from [47].

Figure 3.3: Vitis AI and hardware platforms by Xilinx.

Lastly, an open-source project known as PYNQ [12] aims to run Python on Zynq, Zynq Ultrascale+, Zynq RFSoc and Alveo devices. Nonetheless, there weren't any image files available for the Zybo-Z7 development board before this work. The Pynq environment is shown in figure 3.4. The lowest level uses Vitis, Pynq and user IPs, and provides hardware specification by means of Xilinx Support Archives (*.xsa) or configuration files such as bitstreams. Knowing the target hardware, the Board Support Package (BSP) could then be built. The BSP provides drivers and libraries like the Generic FAT system and the Platform Management API for Zynq [48]. Following this, the BSP could be used as input to the XRT which will support the Linux kernel. The Linux distribution employed is Xilinx's Petalinux. The consecutive layer supplies the PYNQ libraries needed to run Python along with the user overlays. Overlays are implemented through partial reconfiguration of the Programmable Logic (PL). The environment ends with the user application; Jupyter Notebooks running over an IPython kernel.

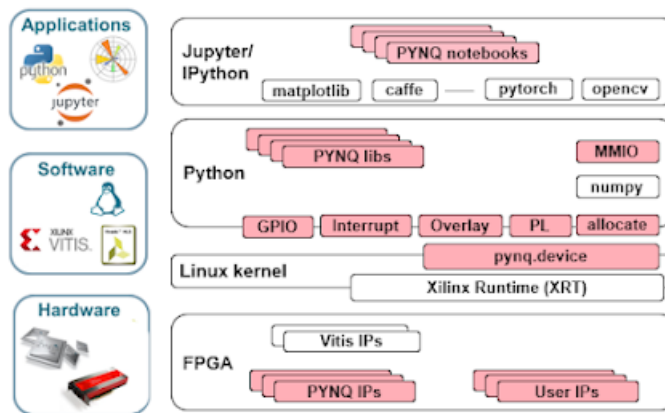


Figure 3.4: Pynq development environment. Taken from [12].

It's worth mentioning that one attempt to improve the capabilities of Pynq is Xilinx's Project "DPU on Pynq" [49]. The project releases an overlay with DPU suitable for Pynq. The release includes example notebooks for training and inference, but it is bound to the expensive Zynq Ultrascale+ FPGAs. Currently, the project only supports three boards: Avnet's Ultra96, Xilinx's ZCU104 and Kria KV260.

Also, the design process of a custom hardware-software (hw-sw) architecture capable of running Pynq on a Zybo-Z7 will be described. The architecture could make Pynq suitable for other lightweight embedded systems. In addition, the Pynq image creation process and guidelines for using ML frameworks will be provided.

2 Creating a Pynq image for Zybo-Z7

In this section, the process of creating a Pynq's image for the Zybo-Z7 board is described. The hardware development includes a minimal processing system configuration through a Tool Command Language (*.tcl) script. In addition, the base overlay will also be defined in a *.tcl file. This overlay will contain the peripherals implemented in the PL and will be connected through the AXI-lite interface to the PS. On the other hand, the software's development will employ a pre-built root fs image to compile custom Pynq 2.7 for the Zybo-Z7.

As the process could help grow the Open-source Pynq project, it has been shared on the official Pynq web site [50]. Following this, the stages will be described in some detail.

2.1 Create the Board Directory

The process starts by cloning the Pynq repository from <https://github.com/Xilinx/PYNQ>. Then, it is needed to create a directory for the Zybo-Z7 board with the structure proposed in [51]. The directory will include all the files needed to build the Pynq image. The created directory can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/Pynq_Zybo-Z7/Zybo-Z7 and should be located inside the boards folder of the Pynq local repository.

† Zybo-Z7/

- base/
 - { notebooks/GPIOs.test.ipynb, etc.
 - vivado/contraints/base.xdc
 - build_bitstream.tcl, build_ip.tcl
 - base.py, base.tcl, makefile, etc.
- notebooks/getting_started/*.ipynb
- packages/boot_leds/
 - { boot.py
 - pre.sh
- petalinux_bsp/
 - { hardware_project { build_bitstream.tcl
 - makefile
 - zyboz7.tcl
 - meta-user/recipes-bsp/ { device-tree/files/system-user.dtsi
 - u-boot/
- Zybo-Z7.spec
- xilinx-zyboz7-2020.2.bsp

The base folder contains the overlay files, such as the hardware design (base.tcl), the hardware constraints (base.xdc), the example GPIOs Jupyter notebook (*.ipynb) and the build scripts. Running its makefile will generate the *base.bit* bitstream. The *packages/boot_leds* directory has a minimal routine to configure the PL with the base overlay and turn on the leds at boot. The board support package (BSP) with the minimal architecture capable of running Petalinux is defined in the *petalinux_bsp* directory. Note that the bsp also includes the Linux device tree and some u-boot parameters. Running its makefile will create the bsp *xilinx-zyboz7-2020.2.bsp*, which should be located as indicated above. At last, the *Zybo-Z7.spec* will point out the architecture, the bsp, the base overlay, and the packages that will be included in the custom Pynq image.

Listing 3.1: Zybo-Z7.spec

```

1 ARCH_Zybo-Z7 := arm
2 BSP_Zybo-Z7 :=
3 #BSP_Zybo-Z7 := xilinx-zyboz7-2020.2.bsp
4 BITSTREAM_Zybo-Z7 := base/base.bit
5 FPGA_MANAGER_Zybo-Z7 := 1
6
7 STAGE4_PACKAGES_Zybo-Z7 := xrt pynq boot_leds ethernet pynq_peripherals

```

2.2 Create the Board Support Package (BSP) (30 min approx.)

The BSP's design will use as a reference the Pynq-Z2 example from the Pynq boards folder. Although the Pynq-Z2 and the Zybo-Z7 have the same FPGA (i.e., xc7Z020-clg400), they present some differences. So, the specifications in [8] should be considered for the Zybo-Z7.

- DDR part number: MT41K256M16 RE-125.
- The PS UART1 is routed to the FTDI instead of the UART0 for the Pynq-Z2.
- The crystal oscillator has a frequency of 33.3333 MHz.

Hence, the *zyboz7.tcl* file was modified to consider the aforementioned. The BSP is shown in figure 3.5 along with its Vivado block design in figure 3.6. The design includes the Zynq Processing System to handle the ARM processor, a DDR ram interface, an interruption vector (IRQ) and some Processor System Resets to handle the overlays' peripherals.

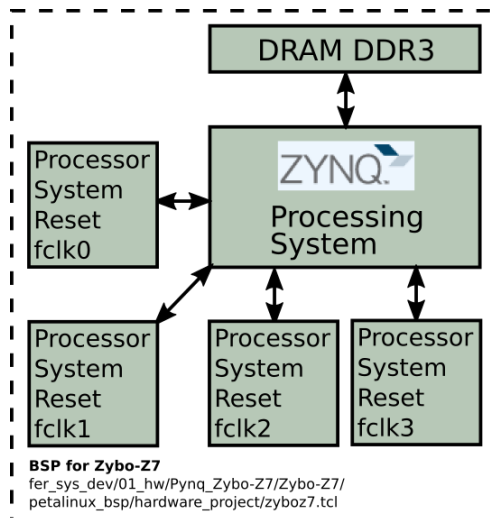


Figure 3.5: Board Support Package adapted for the Zybo-Z7 board.

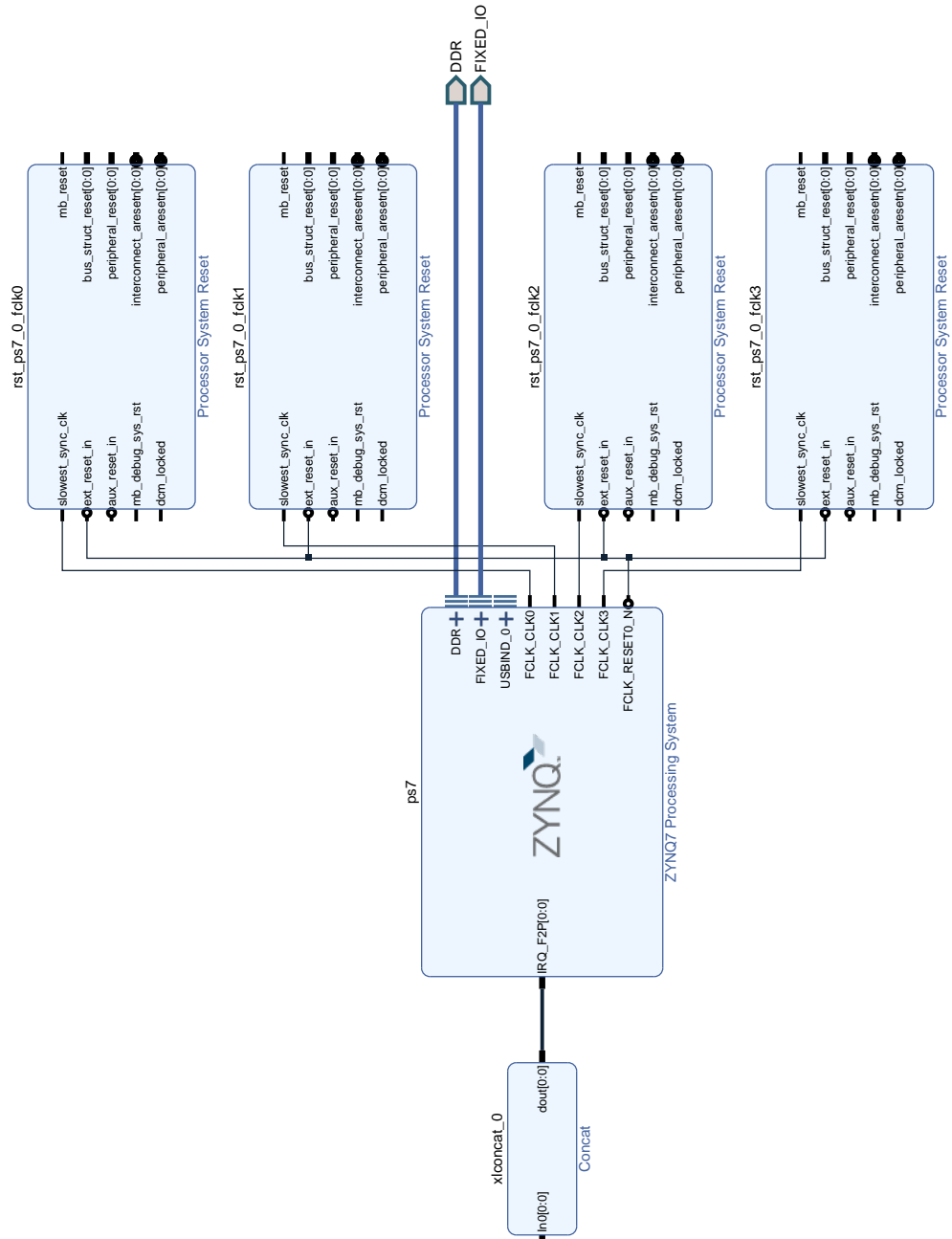


Figure 3.6: Vivado Block design for the BSP targeted to the Zybo-Z7 board.

Listing 3.2 depicts the steps needed to create the bsp.

Listing 3.2: Zybo-Z7.spec.

```
1  $ source /tools/Xilinx/Vitis/2020.2/settings.sh
2  $ source ~/petalinux/2020.2/settings.sh
3  $ petalinux-util --webtalk off
4  Make sure that ~/PYNQ/boards/Zybo-Z7/Zybo-Z7.spec has the line BSP_Zybo-Z7 :=
5  $ cd ~/PYNQ/sdbuild
6  $ make bsp BOARDDIR=/home/user/PYNQ/boards BOARDS="Zybo-Z7"
7  copy the bsp file from ~/PYNQ/sdbuild/output/bsp/Zybo-Z7/xilinx-zyboz7-2020.2.
   bsp to ~/PYNQ/boards/Zybo-Z7/. Thus, along the Zybo-Z7.spec.
8  Add the bsp to the Zybo-Z7.spec: BSP_Zybo := xilinx-zyboz7-2020.2.bsp.
```

2.3 Create the image (2h30 min approx.)

To make the port process simple, the Logictools overlay was removed, and the base overlay was reduced by deleting the Arduino and Raspberry Pi peripherals. Also, the Makefile in the *sdbuild/* folder was also updated to avoid compiling for all the boards (i.e., Pynq-Z1, Pynq-Z2, ZCU-104, sw_repo). The changes were made after line 65, where the Pynq repository is automatically cloned. In addition, it is important to check variations in the *~/PYNQ/build.sh* file. As a reference, the file *Makefile_sdbuild* shared in https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/Pynq_Zybo-Z7/Zybo-Z7) includes the changes made. Moreover, the block diagram for the base overlay is shown in figure 3.7 as well as its Vivado Block design in figure 3.8. The overlay shares the hardware system used for the bsp but adds some IP cores aimed at the PL. These cores include peripherals such as the AXI GPIOs for switches, LEDs, and buttons, along with peripherals like audio codec and HDMI video. Furthermore, the constraints file *base.xdc* was updated employing the Zybo-Z7 master xdc.

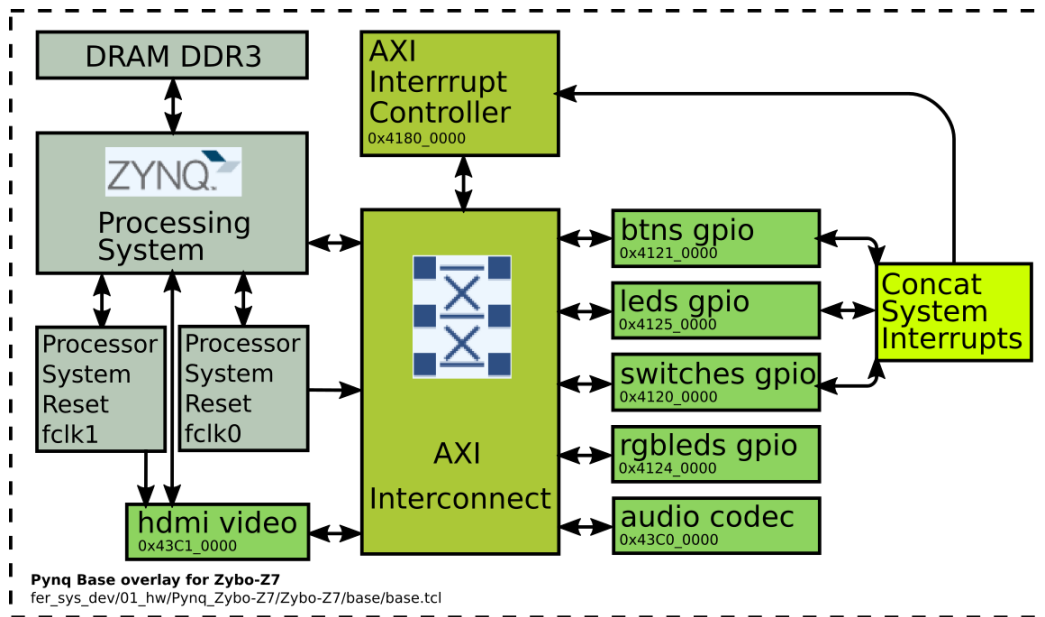


Figure 3.7: Block diagram of base overlay defined in *base.tcl*.

The steps used to create the custom images are described in listing 3.3.

Listing 3.3: Image building.

```

1  Download the PYNQ rootfs arm v2.7 from \url{http://www.pynq.io/board.html}.
2  The downloaded file is named focal.arm.2.7.0_2021_11_17.tar.gz and will be
   placed in ~/PYNQ/sdbuild/.
3  $ cd ~/PYNQ/sdbuild
4  $ make BOARDDIR=/home/user/PYNQ/boards BOARDS="Zybo-Z7" PREBUILT=focal.arm.2.7.0
   _2021_11_17.tar.gz nocheck_images
5  Wait patiently ...

```

When these steps are finished, an image file named *Zybo-Z7-2.7.0.img* will be stored in *~/PYNQ/sdbuild/output* directory. The image size will be around 7.6 GB.

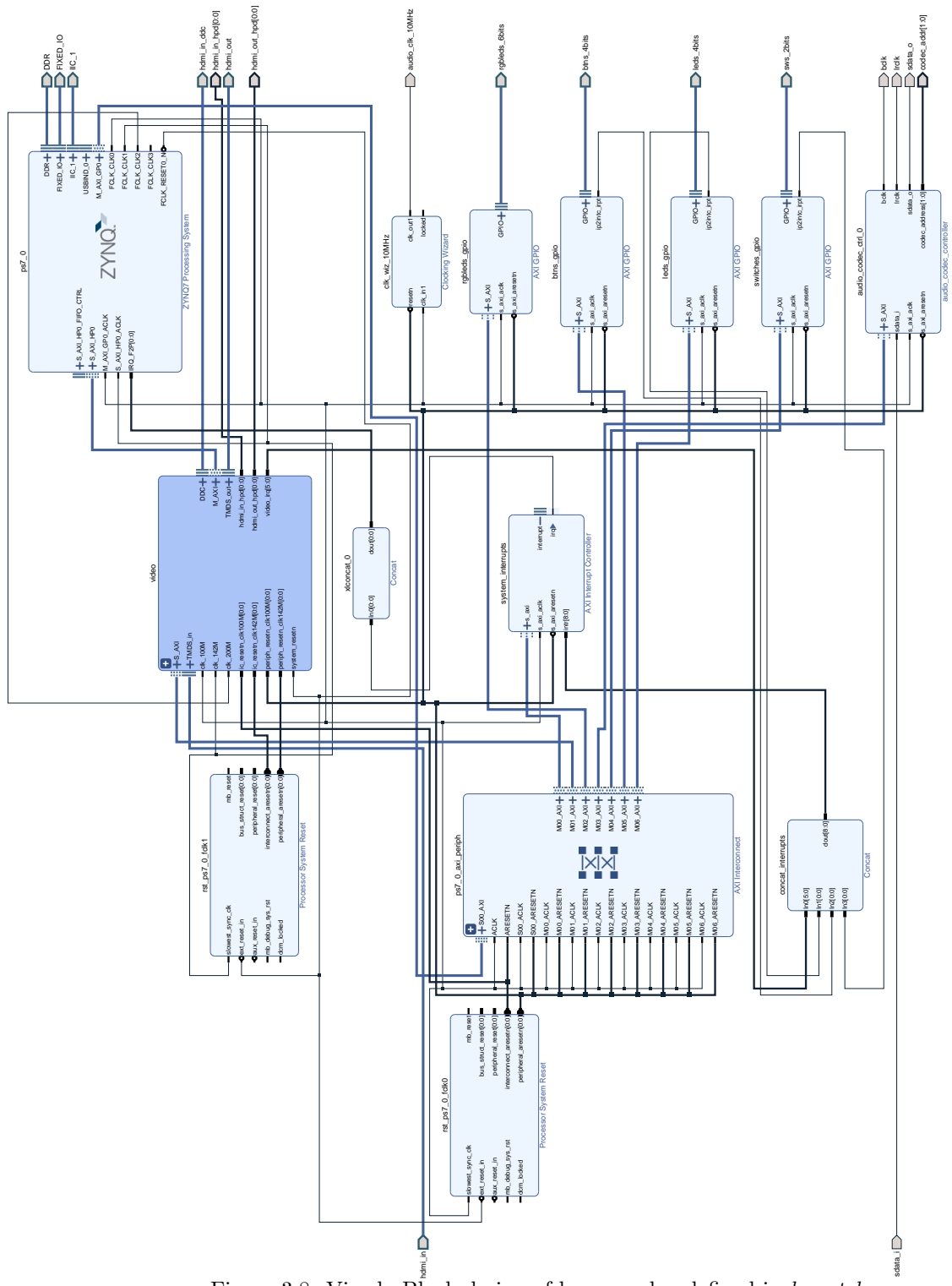


Figure 3.8: Vivado Block design of base overlay defined in `base.tcl`.

2.4 Copy image to micro SD card (20 min approx.)

The final step is to copy the custom Pynq 2.7 image to a micro SD card. In listing 3.4, the commands employed are presented.

Listing 3.4: Copying image to SD card.

```
1 $ cd ~/PYNQ/sdbuild/output
2 $ df -h
3 Select the device corresponding to your SD card. Be careful, you could damage
   your OS if choose wrong.
4 $ umount /dev/sXX1 (for instance, mine was in /dev/sdb1).
5 $ sudo dd bs=4M if=Zybo-Z7-2.7.0.img of=/dev/sXX status=progress
```

Then, insert the SD card into the Zybo-Z7 board and connect it to your local network via Ethernet before powering it on. In the Pynq serial terminal, type `ifconfig` to find out what your local IP address is. Then search for `eth0/inet: 192.168.*.*`. In your internet browser, write the local IP with the port number 9090, for example: `192.168.0.25:9090`. If everything is okay, Pynq will be running and you should be asked to login. Use `xilinx` as the password. For instance, the Figure 3.9 shows Pynq 2.7 running on Zybo-Z7.

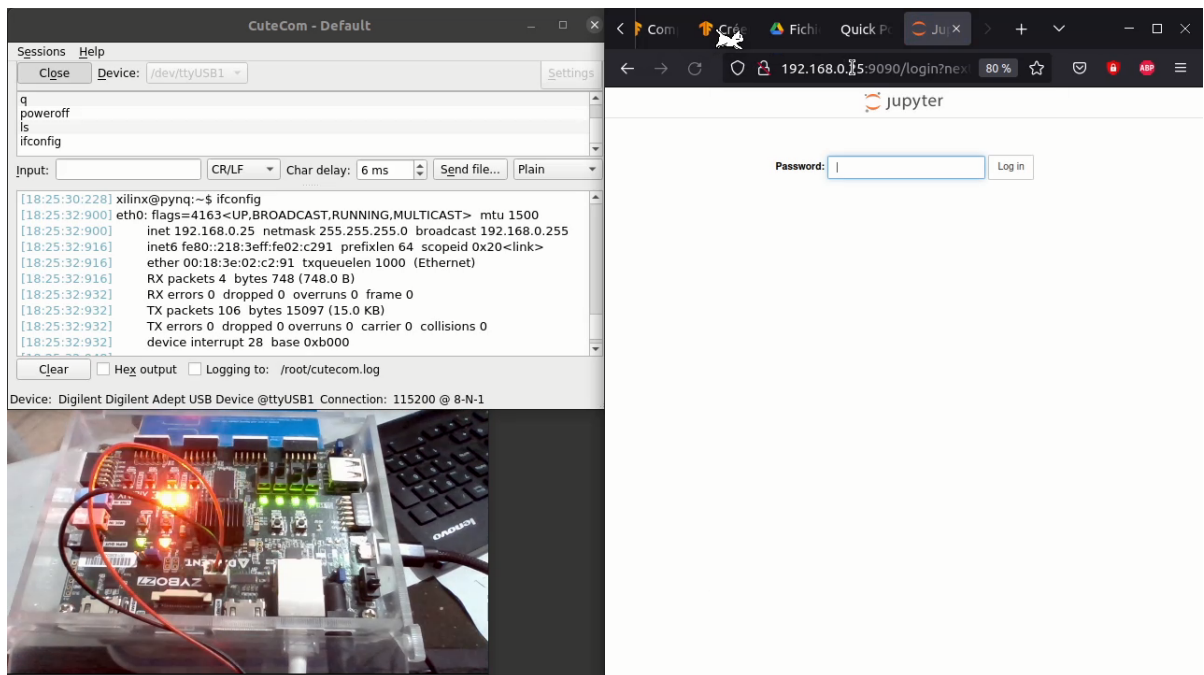


Figure 3.9: Pynq 2.7 running on Zybo-Z7.

Next, to test the base overlay, you could go to `base/GPIOs_test.ipynb` and run the notebook. The custom Pynq 2.7 image created for the Zybo-Z7 can be downloaded from <https://drive.google.com/file/>

d/1kCT_UZjIuDym3yhI739xY1OY1Q4byCHx/. Lastly, a video demonstrating the Pynq 2.7 image running on the board and the base overlay working can be found at <https://www.youtube.com/watch?v=a1FPf9TzUzs>. Next, to test the base overlay you could go to the *base/GPIOs_test.ipynb* and run the notebook. The custom Pynq 2.7 image created for the Zybo-Z7 can be downloaded from https://drive.google.com/file/d/1kCT_UZjIuDym3yhI739xY1OY1Q4byCHx/. Lastly, a video demonstrating the Pynq 2.7 image running on the board and the base overlay working can be found at <https://www.youtube.com/watch?v=a1FPf9TzUzs>.

3 Testing ML Frameworks on the hw-sw architectures proposed

In this section, the process of testing some popular ML frameworks in the hardware-software architecture proposed before is described. The hw-sw architecture will use the Pynq 2.7 image presented in the previous section. Because the work aims to extend the capabilities of training ML algorithms for embedded systems, it has also been shared in the Pynq community. The frameworks tested were Tensorflow and PyTorch and can be found at:

<https://discuss.pynq.io/t/testing-tensorflow-2-5-in-zybo-z7-running-pynq-2-7/4157>
<https://discuss.pynq.io/t/testing-pytorch-1-8-in-zybo-z7-running-pynq-2-7/4181>.

3.1 Tensorflow 2.5 in Zybo-Z7 running Pynq 2.7

3.1.1 Installing Python3.7 (50 min approx.)

The Pynq 2.7 release comes with Python3.8, but the latest TensorFlow wheel available for the armv7l architecture is compiled for Python3.7. For this reason, the former Python version must be installed in Pynq. This can be accomplished by following the commands presented in listing 3.5, which were taken from [52].

Listing 3.5: Installing Python3.7 on Pynq 2.7.

```
1 $ cd /home/xilinx/  
2 $ wget http://www.python.org/ftp/python/3.7.0/Python-3.7.0.tar.xz  
3 $ tar -xf Python-3.7.0.tar.xz  
4 $ cd Python-3.7.0/  
5 $ ./configure  
6 $ make install
```

3.1.2 Create a virtual environment for Python3.7 (5 min approx.)

To avoid unresolved dependencies between packages and be able to create an IPython kernel in the future, it is necessary to create a virtual environment. This process is summarized in listing 3.6.

Listing 3.6: Creating a virtual environment for Python3.7.

```
1 $ cd /home/xilinx/
2 $ python3.7 -m pip install virtualenv
3 $ python3.7 -m virtualenv env
4 $ source /env/bin/activate
```

3.1.3 Install TensorFlow in the virtual environment (> 5h approx.)

Warning: This stage could take a lot of time; for that reason, the estimated time for each command is given. The precompiled TensorFlow 2.5 wheel for the armv7l architecture can be found in [53]. There are two ways to get the wheel: use the Katsuya Hyodo scripts, as shown in listing 3.7, or download it from https://drive.google.com/uc?id=liqylkLsgwHxB_nyZ1H4UmCY3Gy47q1OS and then paste it into `/home/xilinx/` from the SD card's ROOT partion.

Listing 3.7: Hyodo scripts for downloading TF wheel.

```
1 $ wget https://raw.githubusercontent.com/PINTO0309/Tensorflow-bin/main/
   previous_versions/download_tensorflow-2.5.0-cp37-none-linux_armv7l_numpy1195
   .sh 2
2 $ chmod +x download_tensorflow-2.5.0-cp37-none-linux_armv7l_numpy1195.sh
3 $ ./download_tensorflow-2.5.0-cp37-none-linux_armv7l_numpy1195.sh
```

Then, install some of the libraries needed by Tensorflow 2.5 inside the Python3.7 environment. For instance, the H5PY and matplotlib are installed using pip3.7. Other library dependencies would be resolved by pip while TensorFlow is installed (e.g., Numpy, etc.). Listing 3.8 describes the commands employed.

Listing 3.8: Installing TensorFlow 2.5 wheel.

```
1 (env) $ apt update # 2 min approx.
2 (env) $ apt install libhdf5-dev # 2 min approx.
3 (env) $ pip3.7 install --no-binary=h5py h5py # 2 h approx.
4 (env) $ pip3.7 install tensorflow-2.5.0-cp37-none-linuxarmv7l.whl # 2h approx.
5 (env) $ exec $SHELL
6 $ cd /home/xilinx/
7 $ source env/bin/activate
8 (env) $ python3.7 -m pip install matplotlib # 30 min approx.
9 (env) $ python3.7
10 >>> import tensorflow
```

```

11 >>> tensorflow.version
12 The last command should return '2.5.0'.

```

3.1.4 Create the IPython Kernel (1h approx.)

To run TensorFlow in a Jupyter notebook, an IPython kernel must be created. This process is shown in the listing 3.9.

Listing 3.9: IPython kernel with TensorFlow 2.5.

```

1 (env) $ python3.7 -m pip install ipykernel # 1h approx.
2 (env) $ python3.7 -m ipykernel install --user --name=tfenv
3 After this step the kernel for tensorflow has been created.

```

3.1.5 Testing Tensorflow 2.5 running on Zybo-Z7 with Pynq 2.7

The notebook employed to test Tensorflow on the Zybo-Z7 can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/Pynq_Zybo-Z7. It includes the small CNN model shown in figure 3.10. Furthermore, the Mnist data set and the Adam optimizer are used to train the model on the hw-sw architecture.

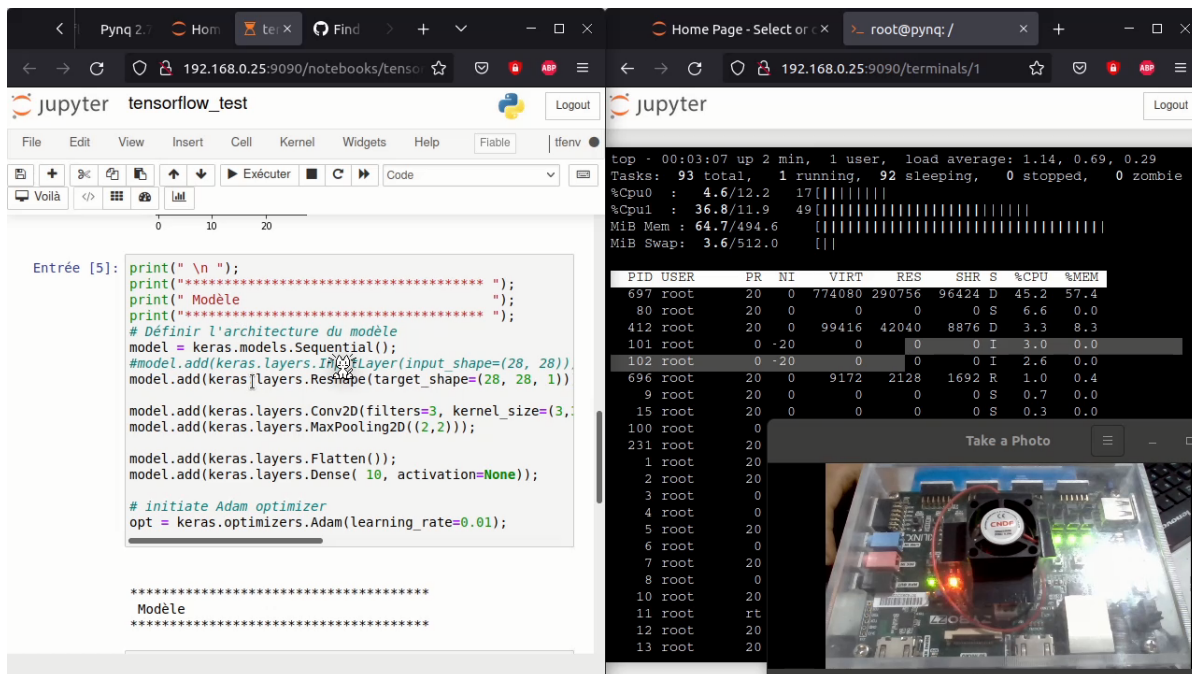


Figure 3.10: Tensorflow running on Zybo-Z7 with Pynq.

However, when the execution calls `model.fit` in the last cell, the kernel returns the error shown in figure 3.11. After some debugging, the error led to a missing case for the `armv7l` in the file `cpu_utils.cc` of the TensorFlow source code. Something similar to the one seen for the `aarch64` in <https://github.com/tensorflow/tensorflow/pull/46643/files/239839b09e02b5766d61041f31027de4eb882c45>. However, the TF `*.cc` files are not available since the wheel is a pre-compiled package. After a quick search using `ls` and `grep`, the only available files are `*.so` libraries and `*.h` header files.

Possible workarounds are: First, cross-compiling Tensorflow from source code, considering both the Python3.8 available in the Pynq 2.7 release and the `armv7l` architecture. Possible workarounds are: First, cross-compiling Tensorflow from source code considering both, the Python3.8 available in the Pynq 2.7 release and the `armv7l` architecture. And second, using FPGAs with ARM 64-bit processors like the Zynq Ultrascale+ family. The last one because there are wheels available for this architecture that have been successfully tested in other platforms (e.g. Raspberry PI 4).

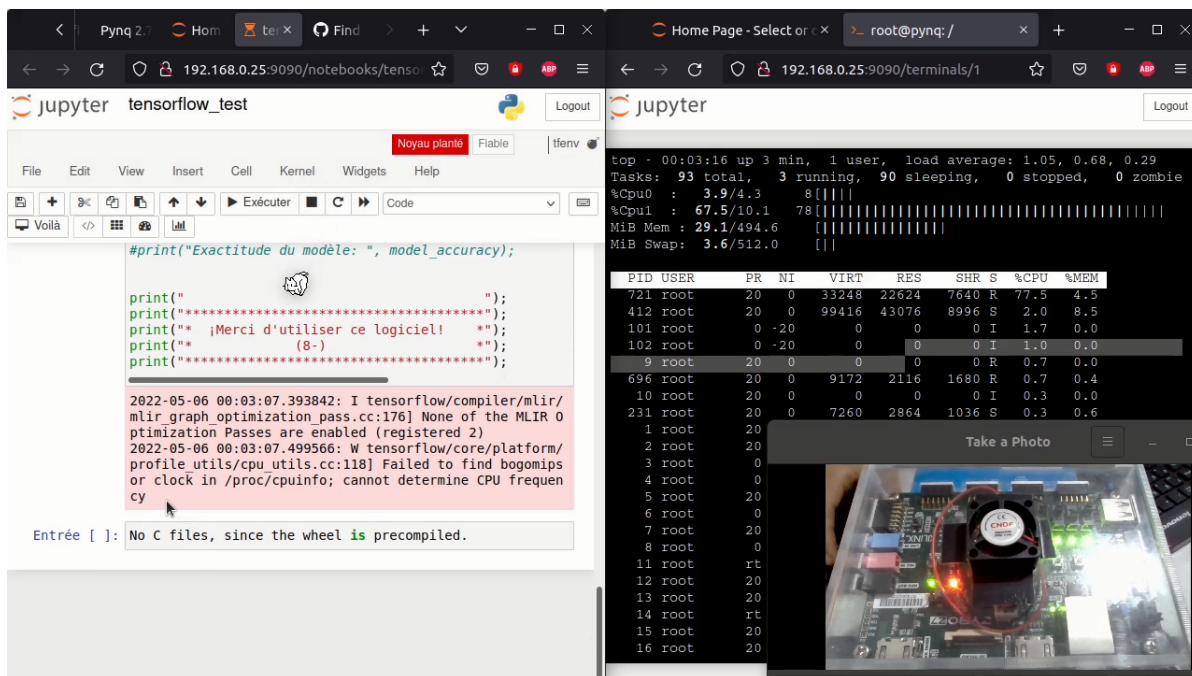


Figure 3.11: TensorFlow `model.fit` error.

The debugging process has been registered in <https://www.youtube.com/watch?v=3m7kavySEYE>.

Lastly, the following should be taken into account before employing this approach:

- The installation time could take more than 5 hours, and during this process, the board must be kept powered and cooled.
- The training process will fail in the `model.fit` step. However, as far as I know, other TensorFlow methods could be executed without concern.
- The inference process could be aimed through loading pre-trained parameters.

- Other frameworks, such as PyTorch 1.8, could be used to train in the same hw-sw architecture. This will be presented in the following section.

3.2 PyTorch 1.8 in Zybo-Z7 running Pynq 2.7

3.2.1 Installing Python3.7 (50 min approx.)

The Pynq 2.7 release comes with Python3.8, but the latest PyTorch wheels found for the armv7l architecture are compiled for Python3.7. For this reason, the former Python version must be installed in Pynq. To install the required version, follow the commands shown in listing 3.10.

Listing 3.10: Installing Python3.7 on Pynq 2.7.

```
1 $ cd /home/xilinx/  
2 $ wget http://www.python.org/ftp/python/3.7.0/Python-3.7.0.tar.xz  
3 $ tar -xf Python-3.7.0.tar.xz  
4 $ cd Python-3.7.0/  
5 $ ./configure  
6 $ make install
```

3.2.2 Create a virtual environment for Python3.7 (5 min approx.)

To avoid unresolved dependencies between packages and be able to create an IPython kernel in the future, it is necessary to create a virtual environment. This can be accomplished with the commands in listing 3.11.

Listing 3.11: Creating IPython kernel for PyTorch 1.8 on Pynq 2.7.

```
1 $ cd /home/xilinx/  
2 $ python3.7 -m pip install virtualenv  
3 $ python3.7 -m virtualenv pytorch_env  
4 $ source pytorch_env/bin/activate
```

3.2.3 Install PyTorch in the virtual environment (> 2h approx.)

PyTorch requires two wheels to be installed: Torch Vision 0.9 and Torch 1.8. These wheels were precompiled by Chia-Wei Wang and can be downloaded from <https://github.com/CW-B-W/PyTorch-and-Vision-for-Raspberry-Pi-4B>. The installation process is described in listing 3.12.

Listing 3.12: Installing PyTorch 1.8

```
1 $ git clone https://github.com/CW-B-W/PyTorch-and-Vision-for-Raspberry-Pi-4B
2 $ cd PyTorch-and-Vision-for-Raspberry-Pi-4B/
3 $ pip3.7 install torch-1.8.0a0+56b43f4-cp37-cp37m-linux_armv7l.whl
4 $ pip3.7 install torchvision-0.9.0a0+8fb5838-cp37-cp37m-linux_armv7l.whl
5 $ python3.7 -m pip install matplotlib
6 $ apt install libopenblas-dev
7 $ exec $SHELL
8 $ cd /home/xilinx
9 $ source pytorch_env/bin/activate
10 $ python3.7
11 >>> import torch
12 >>> print(torch.__version__)
13 The last command should return '1.8.0a0+56b43f4'.
```

3.2.4 Create the IPython Kernel (1h approx.)

To be able to use PyTorch in the Jupyter notebooks, an IPython kernel should be created. Listing 3.13 presents the commands needed.

Listing 3.13: Creating the IPython kernel for PyTorch 1.8

```
1 (pytorch_env) $ python3.7 -m pip install ipykernel # 1h approx.
2 (pytorch_env) $ python3.7 -m ipykernel install --user --name=pytorch_kernel
3 After this step the kernel for PyTorch has been created.
```

3.2.5 Testing PyTorch 1.8 running on Zybo-Z7 with Pynq 2.7

The notebook used to test PyTorch is “A simple PyTorch Training Loop” from the Introduction to PyTorch found at https://www.youtube.com/watch?v=IC0_FriX-sw. The test includes a CNN model with two convolutional layers trained with CIFAR10 and the SGD optimizer. The training process takes about one hour; about five minutes per batch of 2000 images. The SoC memory is around 495 MB, of which up to 85 % was used. Table 3.1 shows the loss values obtained for a laptop and the custom hardware-software architecture proposed along with the error percentage. It is seen that loss values are similar for the two platforms. The accuracy obtained on both platforms is around 54%, however, as the loss tendency shows, accuracy could be improved by increasing the number of epochs. Unfortunately, this will also increase the training time at a rate of approximately 30 minutes per epoch.

Epoch	Samples	hw-sw arch.	Laptop	% Error
1	2000	2.189	2.235	2.058
1	4000	1.836	1.940	5.360
1	6000	1.673	1.713	2.335
1	8000	1.587	1.573	0.889
1	10000	1.540	1.507	2.189
1	12000	1.450	1.442	0.554
2	2000	1.401	1.378	1.669
2	4000	1.367	1.364	0.219
2	6000	1.348	1.349	0.074
2	8000	1.324	1.319	0.379
2	10000	1.278	1.284	0.480
2	12000	1.261	1.267	0.473

Table 3.1: Loss values for laptop and the hw-sw architecture proposed.

In figure 3.12 the CNN model trained with PyTorch in the Zybo-Z7 is shown. In addition, a copy of the wheels and notebooks employed can be found at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/Pynq_Zybo-Z7.

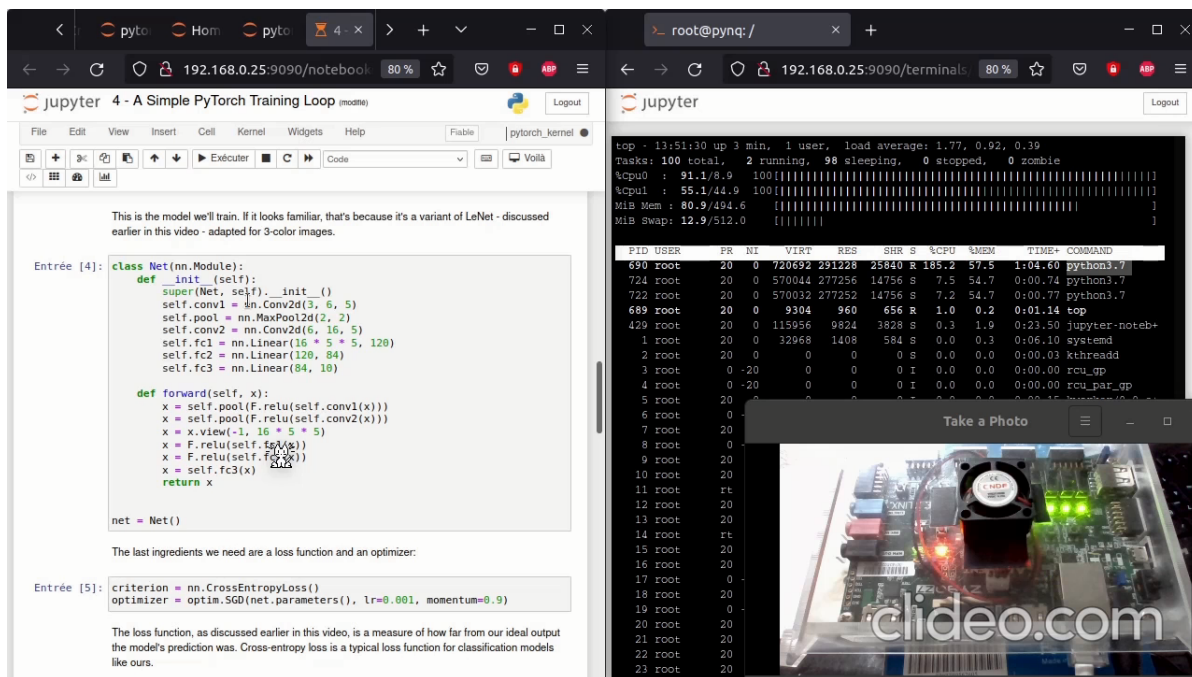


Figure 3.12: PyTorch 1.8 running on Zybo-Z7 with Pynq.

3.3 Testing model M6 in Zybo-Z7 running Pynq 2.7

The hw-sw architecture was also used to train the model M6. As TensorFlow presented some troubles (see subsection 3.1), the model was mapped to PyTorch. In addition, as each ML framework implements a custom version of layers and optimizers, it is expected to achieve different accuracy values. The notebook created can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/Pynq_Zybo-Z7/FER_M6_pytorch/M6_jaffe_6emo_pytorch.ipynb, and in appendix B.1. The algorithm 18 presents the PyTorch class needed to define the model. Here, methods are used to describe its behavior for backward and forward propagation.

Algorithm 18 Model M6 class for PyTorch.

```
1: class M6FER_Net(nn.Module) :
2:   def __init__(self) :
3:     super(M6FER_Net, self).__init__();
4:     self.conv1 = nn.Conv2d( 1, 32, 5);
5:     self.conv2 = nn.Conv2d(32, 64, 5);
6:     self.conv3 = nn.Conv2d(64, 128, 5);
7:     self.mpool = nn.MaxPool2d(2, 2);
8:     self.fc1 = nn.Linear(128 * 4 * 4, 6);
9:     * **
10:  def forward(self, x) :
11:    x = self.mpool(F.relu(self.conv1(x)));
12:    x = self.mpool(F.relu(self.conv2(x)));
13:    x = self.mpool(F.relu(self.conv3(x)));
14:    x = x.view(-1, 128 * 4 * 4);
15:    x = self.fc1(x);
16:    return x;
17: end class
```

Besides, Figure 3.13 depicts the setting employed, which was tested on the Zybo-Z7 board with Pynq 2.7 and PyTorch 1.8 installed. The testing process can be found at https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/Pynq_Zybo-Z7/FER_M6_pytorch/FER_M6_pytorch_Zyboz7_20220722.mp4. While the board has 1 GB of RAM, the Pynq image mapped 495 MB of it, from which up to 80 % was used. Moreover, although the test was designed for the M6 model, the architecture could work with other similar CNN models.

Furthermore, table 3.2 presents the time spent training the M6 model in the Zybo-Z7 and on a laptop. On the Zybo-Z7, the training process took 1h50; which is about 10 minutes per batch of 500 images, yielding a 44 % accuracy. Therefore, the hw-sw architecture proposed was slower than the laptop, which spent 50 s and got 50 % of accuracy. However, this gap could be reduced with more capable FPGAs. For instance, the Zynq Ultrascale+ family provides faster 64-bit MPSoC suitable for custom IP cores targeting CNN's training (e.g., DPU).

Model M6 in PyTorch



HW-SW architecture for training model M6 with PyTorch 1.8 in Pynq 2.7

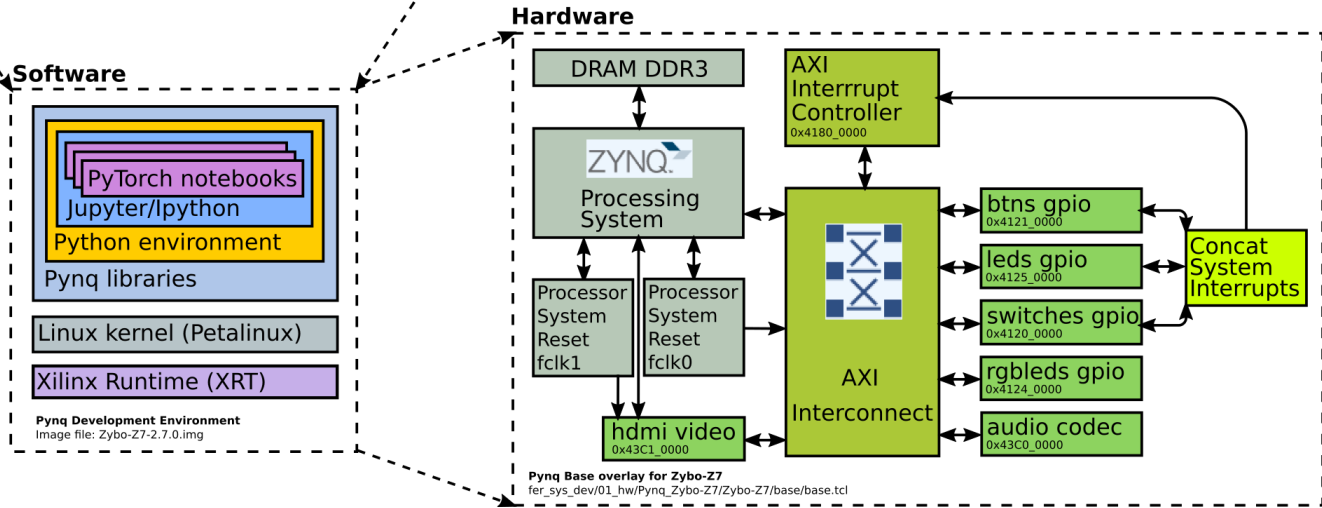


Figure 3.13: Hw-sw architecture for training the model M6 with PyTorch 1.8 in Pynq 2.7.

Platform Name	Hardware specs	Software	Accuracy	Execution time
Legion 5	Processor: AMD Ryzen 7 4800H RAM: 16 GB GPU: Nvidia GeForce GTX 1660 Ti.	OS: Ubuntu 20.04.2 LTS kernel: 5.13.0-37-generic library: PyTorch 1.8. application: M6***.ipynb	50 %	60 s
Zybo-Z7	FPGA: Xilinx Zynq XC7Z020. Processor: ARM A9 core at 660 MHz RAM: 1 GB DDR3.	OS: Pynq 2.7 kernel: Petalinux library: PyTorch 1.8 application: M6***.ipynb	44 %	1h50

Table 3.2: M6 training performance comparison.

4 Resiliency: A Framework for Training a CNN with a custom Hardware-Software Architecture.

Employing custom hw-sw architectures for training and running the inference of CNNs may broaden the scope of the ML applications. However, it is an overwhelming process that requires deep knowledge of FPGA’s design. Despite this, several works that aim to accelerate CNNs on FPGAs can be found at the literature. For instance, in [54] authors proposed the Argus framework for utilizing Convolutional Layer Processors (CLP) in large and expensive Virtex-7 (i.e., 485T and 690T) FPGAs. Argus takes the CNN model description and provides the RTL description of the architecture. Another example is the ReCon framework presented in [55]. Hence, the authors aim to accelerate a Hybrid Semantic Segmentation application. ReCon employs Zynq UltraScale+ MPSoC FPGAs with the partial reconfiguration technique. In addition, [56] introduces TF2FPGA, a framework that aims to map TensorFlow CNN models to Zynq FPGAs. Unfortunately, the work lacks detail about the hw-sw architecture proposed. Besides, [16] presents a Task assignment framework for acceleration of CNNs on Zynq FPGAs using Deep Processing Units (DPUs). The framework targeted the Ultrascale+ ZCU104 and studied the effects of thread scheduling on the execution time. Lastly, the OMNI framework described in [15] integrates hardware and software optimizations for sparse CNNs tested on the ZCU102 and ZCU706 development boards. In OMNI the authors analyzed the computation dataflow and the best suitable Processing Elements (PE) design.

Nonetheless, these works do not provide access to relevant files or repositories for reproducing their results. Furthermore, efforts towards training CNNs on FPGAs with hw-sw architectures were not found in literature. As a consequence, one of the goals of this section is to fill these gaps. The section presents a framework for CNN’s execution in custom hardware-software architectures called Resiliency. It has been used to train and run the inference of the single-channel M6 model, a CNN-based FER system presented in Chapter 1. Moreover, algorithm 19 describes the framework design flow, and Figure 3.14 gives an overview.

To begin with, the user must provide the CNN model, the preprocessed data set, and an APSoC development board. To run the inference, the CNN model needs to be trained with TensorFlow and quantized with tflite. The parameters obtained for each layer must be copied to the board’s microSD card. The FER SoC is described by the *.xsa file found in https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/fer_soc_bd_wrapper.xsa. Also, if needed, the user could change the design by using the Vivado project provided at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/fer_soc. Besides, the Vitis project provides the functions and drivers required to handle the custom IP cores in the FER SoC. The Vitis project is located at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/fer_soc/firmware.

The model architecture should be mapped to an application *.c file (e.g. M6FER.c in https://gitlab.com/dorfell/fer_sys_dev/-/blob/master/01_hw/fer_soc/firmware/app0/src/M6FER.c). The layers supported include the tflite version of convolution, maxpooling and fully connected, but the user could add its own IP cores to the SoC. The core files are also available at https://gitlab.com/dorfell/fer_sys_dev/-/tree/master/01_hw/ip_repo, along with some simulation template files in the folders *conv.1.0/src* and *tflite_mbgm.1.0/src*. Before proceeding, check paths to the data and layer parameters, as well as sizes.

Once the Vitis project compiles, the board can be configured. The user can then control the execution flow using a serial terminal from another computer, or the inference can be run automatically by adjusting the main application loop. Before proceeding, check paths to the data and layer parameters as well as sizes. Once the Vitis project compiles the board can be configured. Next, the user can control the execution flow by using a serial terminal from another computer or run the inference automatically adjusting the application main loop.

For training a CNN model, the following is needed: the Pynq 2.7 image copied to the microSD card and the model in a Jupyter notebook. The Pynq image can be downloaded from https://drive.google.com/file/d/1kCT_UZjIuDym3yhI739xY1OY1Q4byCHx/. However, although TensorFlow and PyTorch could be installed, TensorFlow has a bug in the model.fit step. For that reason, PyTorch should be used instead. One workaround to use TensorFlow is to change to a 64-bit MPSoC [57], because that wheel has been successfully tested on the Raspberry PI 4 and they both use similar ARM CPUs. Next, turn on the board and login to the Jupyter server at 192.168.0.X:9090. Make sure you've installed the ML framework following the instructions given in section 3. Then, run the training, taking into account that this step could take several hours because of the embedded system specs: a 32-bit CPU at 600 MHz, RAM 495 MB, etc. At last, to assess performance, the following metrics can be utilized: execution time, accuracy, and loss.

Algorithm 19 Framework for FER systems in a custom HW-SW architecture.

```

1: Require:
2:   Single channel CNN.
3:   Pre – processed data set.
4:   APSoC development board.
5:   if (inference) then
6:     Ensure:
7:       CNN model trained and quantized.
8:       Model parameters in SDCard.
9:       Vivado project :
10:      Hardware files (*.xsa, *.bitstream).
11:      Vitis project :
12:      * .c application to define model.
13:     Execute:
14:       Compile project.
15:       Program FPGA.
16:       Open Serial terminal.
17:       Run inference.
18:   Else if (train) then
19:     Ensure:
20:       CNN model in Jupyter notebook
21:       Pynq image and notebook in SDCard
22:     Execute:
23:       Start Jupyter server
24:       Install ML frameworks
25:       Run training
26:   End if
27: Assessment:
28:   Accuracy, loss, execution time, etc.

```

RESILIENCY: FRAMEWORK FOR FER SYSTEMS IN CUSTOM HW-SW ARCHITECTURES

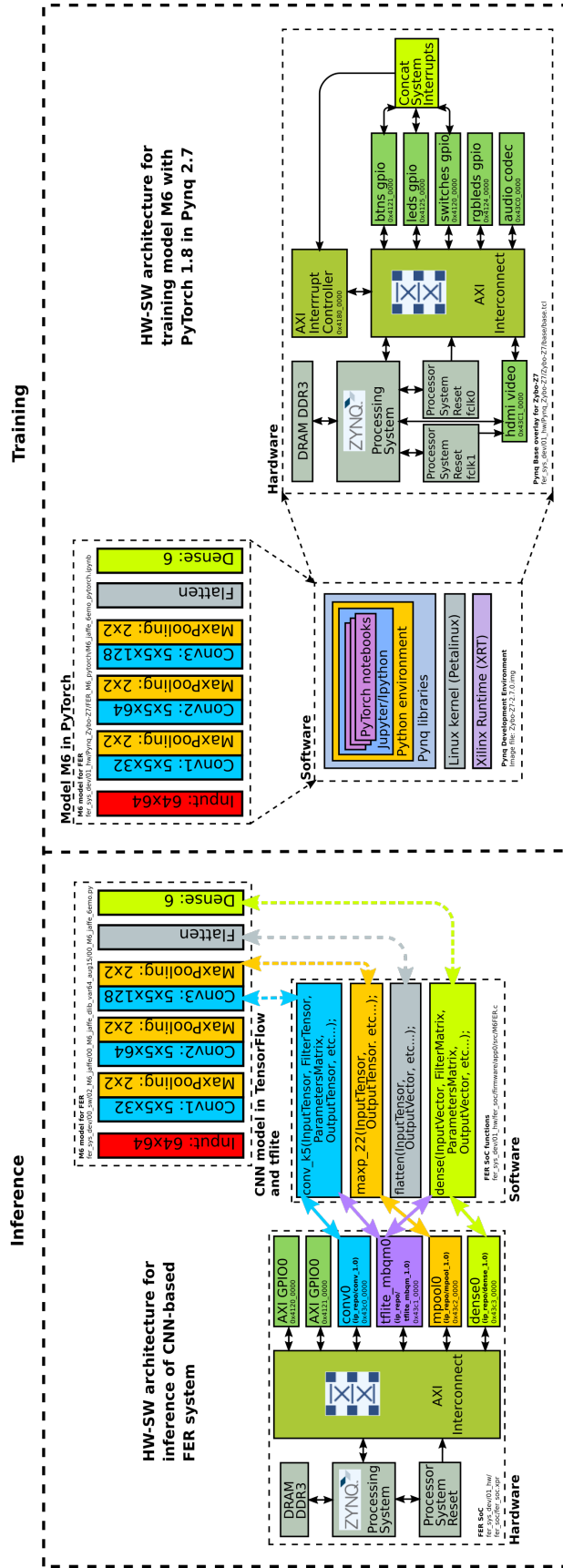


Figure 3.14: Resiliency framework overview.

Furthermore, the custom hw-sw architectures involved in the framework could be modified to improve the performance metrics. Some of the tips the user should keep in mind are:

- The training process is affected by the APSoC operation frequency (i.e., 660 *MHz* for Zybo-Z7) and the RAM memory available. The execution time could be reduced by using a Multiprocessor System on Chip (MPSoC) instead [57]. For instance, MPSoCs employ ARM Cortex-A53 processors that can run up to 2.3 *GHz*.
- Another way to decrease the training time is by adding Deep Processing Units (DPU) [49] to the hw-sw architecture. They can, however, only be used with some Ultrascale+ boards that support MPSoCs, according to [49].
- As the partial results of each layer are kept for computing the next one, complex models usually entail big stacks. Hence, when targeting small embedded systems, the user should consider single-channel CNNs over multichannel models. This behavior applies to the inference and training processes.
- The FER SoC could obtain acceptable accuracy and inference time when working with integers representation. Therefore, the CNN models should be quantized first to determine whether mapping the model to the hw-sw architecture.
- The throughput of the FER SoC proposed for inference could be increased with more instances per custom IP core. Here the FPGA resources for the Place and Route stages should also be considered. These could be closely monitored in the Vivado Open Device and Utilization Report.

5 Discussion

Custom hw-sw architectures capable of executing the training stage of ML algorithms broaden the application scope for embedded systems. This also includes the use of advanced techniques such as Transfer Learning [22]. However, the architecture's performance depends on several factors, like data set size, number of model layers, SoC memory, and operation frequency, among others. Therefore, a trade-off between the training time and the platform capabilities will always be present.

Furthermore, traditional ML frameworks are best suited for multi-core CPUs or GPUs because their core operations usually involve high-precision data representation and could be made in parallel. Nonetheless, embedded systems CPUs are limited to up to 4 threads in the best-case scenario, while GPUs, although having hundreds of threads, are more expensive and have higher power consumption.

On the other hand, Zynq FPGAs integrate ARM CPUs with Programmable Logic (PL), which could help to increase the number of operations executed in parallel by employing several instances of specific designed IP cores. However, to obtain an acceptable performance in training, larger FPGAs with MPSoCs should be used (e.g., the Ultrascale+ family). MPSoCs include multi-core 64-bit CPUs operating up to 2.4 *GHz*. And Ultrascale FPGAs have enough PL area for placing and routing several IP cores for deep learning operations, such as DPUs.

Finally, while there is no general agreement on how frameworks should accelerate CNNs on hardware, the literature shows a preference for MPSoCs. Additionally, the shared methodology between the APSoC used in this work and MPSoCs points to the potential of using them with Pynq and DPUs, which would reduce the complexity of training CNNs on custom hw-sw architectures while maintaining acceptable performance.

6 Conclusion

This chapter presents a custom hw-sw architecture able to run Pynq 2.7 on the Zybo-Z7, as well as the BSP with the minimal hardware design needed. Moreover, the base overlay that includes some AXI IP cores implemented in the PL is also described. As mentioned, Pynq is based on Petalinux and its custom image was built taking into account the BSP address map and the drivers and libraries needed to handle the peripherals from Python. Furthermore, the image allows the execution of a Jupyter server on the development board. Besides, the process of preparing Pynq for the ML frameworks was also presented. Furthermore, TensorFlow and PyTorch were tested with simple CNN models trained with the data sets MNIST and CIFAR10. It was found that the TensorFlow wheel package available for the 32-bit ARM architecture has a bug when calling the fit method. Specifically, the method couldn't recognize the CPU frequency needed to launch the optimization. However, it is known that TensorFlow is able to run on ARM 64-bit platforms such as the Raspberry PI 4. PyTorch was also tested by training a CNN model. The loss values obtained were similar to the ones from the laptop execution. Nonetheless, due to the fact that the hw-sw architecture proposed has limited resources such as memory, number of threads, and CPU frequency, the time spent per epoch is slow, around 30 minutes per epoch. Likewise, the M6 model used for FER applications was mapped to PyTorch and tested on the board. Hence, the training time was around 2 hours for the embedded system and 1 minute for the 16-core laptop. Also, the hw-sw architecture was 6% less accurate than the laptop.

On the other hand, the chapter introduces Resiliency. A framework for training and running the inference of CNNs-based FER systems in custom hw-sw architectures. Resiliency provides HDL files, the Vivado and Vitis projects, and the drivers and functions to handle the custom IP cores with their simulation testbenches. Also available are the Pynq image and the notebooks employed when testing TensorFlow and PyTorch. Also, the Pynq image and the notebooks employed when testing TensorFlow and PyTorch are also available. As revealed in the literature and from metrics obtained with the framework, the best alternative is employing FPGAs with MPSoCs.

At last, the chapter provides links to all the files needed to reproduce the results. These files could be used to develop and test overlays with IP cores designed to handle tensor operations, convolutional layers, etc. This paves the way for overlays that extend the capabilities of the proposed hw-sw architecture for lightweight embedded systems based on Zynq devices.

Final Discussion

This work described a design methodology for a single-channel CNN-based FER system, along with the data set preprocessing, LBP and DA. By following this methodology, the model M6 was trained using TensorFlow with the JAFFE data set, achieving an accuracy of 94%. However, the size of the data set and the model make training techniques like EarlyStopping and ReduceLROnPlateau not suitable. Also, a future step to improving the model's performance would be training with larger data sets like CK+ and using cross-validation.

Then, to run the inference of the M6 model on hardware, a custom hardware-software architecture (a.k.a. FER SoC) was introduced. The model compression was made with quantization instead of pruning, since the former would be too aggressive. The library used was tflite from TensorFlow and the quantized model accuracy was 83.33%, with parameters represented by integer numbers. Currently, the FER SoC is capable of computing up to 64 operations at the same time, but thanks to its flexibility, other IP cores' instances could be added as far as the FPGA's available resources allow it.

Next, the work aimed at training a CNN model on a custom hardware-software architecture. The architecture included a BSP capable of running Pynq 2.7 on the Zybo-Z7, along with a base overlay for AXI IP cores. The architecture runs a Jupyter server used to install and test the PyTorch and TensorFlow frameworks with the CIFAR10, JAFFE and MNIST data sets. However, the TensorFlow wheel available for the 32-bit ARM compatible with Pynq 2.7 didn't allow training. Hence, M6 was mapped to PyTorch and the training time was around two hours, whereas one minute for a 16-core laptop. The hw-sw architecture was 6% less accurate than the laptop. In subsequent implementations, the performance could be improved by using faster MPSoCs FPGAs with 64-bit processors.

Afterwards, the framework Resiliency was introduced. It gives the guidelines to train and run the inference of CNN models on an embedded system, and provides the custom hardware-software architectures used in this work. Moreover, the source code is open and available, and the designs are scalable, which facilitates the use of larger and faster devices suitable for DPUs units.

Lastly, the academic production is listed below:

- D. Parra and C. Camargo, "A Systematic Literature Review of Hardware Neural Networks," 2018 IEEE 1st Colombian Conference on Applications in Computational Intelligence (ColCACI), 2018, pp. 1-6, doi: 10.1109/ColCACI.2018.8484858.
- D. Parra and C. Camargo, "Design Methodology for CNN-based FER systems," UNPUBLISHED: ICICT 2023.
- D. Parra, "Pynq 2.7 for Zybo-Z7", 2022. [Online]. Available: <https://discuss.pynq.io/t/pynq-2-7-for-zybo-z7/4124>
- D. Parra, "Testing Tensorflow 2.5 in Zybo-Z7 running Pynq 2.7", 2022. [Online]. Available: <https://discuss.pynq.io/t/testing-tensorflow-2-5-in-zybo-z7-running-pynq-2-7/4157>

- D. Parra, “Testing PyTorch 1.8 in Zybo-Z7 running Pynq 2.7”, 2022. [Online]. Available: <https://discuss.pynq.io/t/testing-pytorch-1-8-in-zybo-z7-running-pynq-2-7/4181>
- D. Parra, “Développement d’un système pour applications FER”, 2022. [Online]. Available: https://gitlab.com/dorfell/fer_sys_dev/

Bibliography

- [1] Y. Tian, T. Kanade, and J. Cohn, "Recognizing action units for facial expression analysis," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 23, no. 2, pp. 97–115, 2001.
- [2] M. Lyons, M. Kamachi, and Gyoba, "Coding facial expressions with gabor wavelets (ivc special issue)," *Modified version of a conference article, that was invited for publication in a special issue of Image and Vision Computing dedicated to a selection of articles from the IEEE Face and Gesture 1998 conference. The special issue never materialized.*, 2020. [Online]. Available: <https://zenodo.org/record/4029680>
- [3] K. Anas, "Facial expression recognition in jaffe database," <https://github.com/anas-899/facial-expression-recognition-Jaffe>, last accessed 12 Dec 2021.
- [4] S. Christos, T. Georgios, Z. Stefanos, and P. Maja, "300 faces in-the-wild challenge: The first facial landmark localization challenge," *Proceedings of IEEE Int'l Conf. on Computer Vision (ICCV-W)*, 2013. [Online]. Available: <https://ibug.doc.ic.ac.uk/resources/facial-point-annotations/>
- [5] B. Yang, J. Cao, R. Ni, and Y. Zhang, "Facial expression recognition using weighted mixture deep neural network based on double-channel facial images," *IEEE Access*, vol. 6, pp. 4630–4640, 2018.
- [6] J. Kim, B. Kim, P. Roy, and D. Jeong, "Efficient facial expression recognition algorithm based on hierarchical deep neural network structure," *IEEE Access*, vol. 7, pp. 41 273–41 285, 2019.
- [7] Digilent, "Zybo z7," <https://digilent.com/reference/programmable-logic/zybo-z7/start>, last accessed 28 Jan 2022.
- [8] —, "Zybo z7 reference manual," <https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual>, last accessed 28 Jan 2022.
- [9] Xilinx, "7 series fpgas configurable logic block," https://www.xilinx.com/support/documentation/user_guides/ug474.7Series_CLB.pdf, last accessed 28 Jan 2022.
- [10] —, "7 series fpgas memory resources," https://www.xilinx.com/support/documentation/user_guides/ug473.7Series_Memory_Resources.pdf, last accessed 28 Jan 2022.
- [11] —, "7 series dsp48e1 slice," https://www.xilinx.com/support/documentation/user_guides/ug479.7Series_DSP48E1.pdf, last accessed 28 Jan 2022.

- [12] AMD-Xilinx, “Pynq: Python productivity,” <http://www.pynq.io/>, last accessed 19 May 2022.
- [13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” *Google Inc.*, pp. 1–14, December 2017.
- [14] V. Bettadapura, “Face expression recognition and analysis: The state of the art,” *CoRR*, vol. abs/1203.6722, pp. 1–27, 2012.
- [15] Y. Liang, L. Lu, and J. Xie, “Omni: A framework for integrating hardware and software optimizations for sparse cnns,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1648–1661, 2021.
- [16] J. Zhu, L. Wang, H. Liu, S. Tian, Q. Deng, and J. Li, “An efficient task assignment framework to accelerate dpu-based convolutional neural network inference on fpgas,” *IEEE Access*, vol. 8, pp. 83 224–83 237, 2020.
- [17] Xie, Siyue, and H. Hu, “Facial expression recognition using hierarchical features with deep comprehensive multipatches aggregation convolutional neural networks,” *IEEE Transactions on Multimedia*, vol. 21, no. 1, pp. 211–220, 2019.
- [18] S. Li and W. Deng, “Deep facial expression recognition: A survey,” *IEEE Transactions on Affective Computing*, vol. 13, no. 3, pp. 1195–1215, 2022.
- [19] “Tensorflow: An open-source software library for machine intelligence,” <https://www.tensorflow.org/>, last accessed 15 Feb 2018.
- [20] tensorflow, “Quantization aware training,” <https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html>, last accessed 28 Jan 2022.
- [21] M. Merenda, C. Porcaro, and D. Iero, “Edge machine learning for ai-enabled iot devices: A review,” *Sensors*, vol. 20, no. 9, 2020. [Online]. Available: <https://www.mdpi.com/1424-8220/20/9/2533>
- [22] TensorFlow, “Transfer learning with tensorflow hub,” https://www.tensorflow.org/tutorials/images/transfer_learning_with_hub, last accessed 17 May 2022.
- [23] M. Lyons, M. Kamachi, and J. Gyoba, “The japanese female facial expression (jaffe) dataset,” <https://doi.org/10.5281/zenodo.3451524>, last accessed 06 Dec 2021.
- [24] M. Z. Uddin, W. Khaksar, and J. Torresen, “Facial expression recognition using salient features and convolutional neural,” *IEEE Access*, vol. 5, pp. 26 146–26 161, 2017.
- [25] C. Zhang, P. Wang, K. Chen, and J. Kamarainen, “Identity-aware convolutional neural networks for facial expression recognition,” *Journal of Systems Engineering and Electronics*, vol. 28, no. 4, pp. 784–792, 2017.
- [26] P. Lucey, J. F. Cohn, T. Kanade, J. Saragih, Z. Ambadar, and I. Matthews, “The extended cohn-kanade dataset (ck+): A complete dataset for action unit and emotion-specified expression,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pp. 94–101, 2010.

- [27] V. Paul and J. Michael J., “Robus real-time object detection,” *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, December 2004.
- [28] K. David, “Dlib-models,” <https://github.com/davisking/dlib-models/>, last accessed 12 Dec 2021.
- [29] K. Davis E., “Dlib-ml: A machine learning toolkit,” *Journal of Machine Learning Research*, vol. 10, no. 2, pp. 1755–1758, 2009.
- [30] Itseez, “Open source computer vision library,” <https://github.com/itseez/opencv>, last accessed 12 Dec 2021.
- [31] S. van der Walt, J. L. Schonberger, J. , Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Goullart, T. Yu, and the scikit-image contributors, “scikit-image: image processing in Python,” *PeerJ*, vol. 2, p. e453, 6 2014. [Online]. Available: <https://doi.org/10.7717/peerj.453>
- [32] skimage, “local_binary_pattern,” https://scikit-image.org/docs/stable/api/skimage.feature.html?highlight=local_binary_pattern#skimage.feature.local_binary_pattern, last accessed 17 Dec 2021.
- [33] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, “Albumentations: Fast and flexible image augmentations,” *Information*, vol. 11, no. 2, 6 2020. [Online]. Available: <https://www.mdpi.com/2078-2489/11/2/125>
- [34] Keras, “Earllystopping,” https://keras.io/api/callbacks/early_stopping/, last accessed 06 April 2023.
- [35] —, “Reduclronplateau,” https://keras.io/api/callbacks/reduce_lr_on_plateau/, last accessed 06 April 2023.
- [36] TensorFlow, “Pruning comprehensive guide,” https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide, last accessed 08 April 2023.
- [37] tensorflow, “Tensorflow lite 8-bit quantization specification,” https://www.tensorflow.org/lite/performance/quantization_spec, last accessed 28 Jan 2022.
- [38] Xilinx, “Field programmable gate array (fpga),” <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>, last accessed 28 Jan 2022.
- [39] Amazon, “Alexa,” <https://www.amazon.com/b?node=21576558011>, last accessed 17 May 2022.
- [40] Google, “Hey google,” <https://assistant.google.com/>, last accessed 17 May 2022.
- [41] Anki, “Vector by anki: A giant roll forward for robot kind,” <https://www.kickstarter.com/projects/anki/vector-by-anki-a-giant-roll-forward-for-robot-kind>, last accessed 17 May 2022.
- [42] D. D. Labs, “Vector 2.0,” <https://www.digitaldreamlabs.com/products/vector-robot>, last accessed 17 May 2022.
- [43] L. AI, “Emo: The coolest ai desktop pet with personality and ideas,” <https://living.ai/emo/>, last accessed 17 May 2022.
- [44] Google, “Coral,” <https://coral.ai/>, last accessed 17 May 2022.

- [45] Intel, “Intel neural compute stick 2 (intel ncs2),” <https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html>, last accessed 17 May 2022.
- [46] AMD-Xilinx, “Vitis ai,” <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>, last accessed 19 May 2022.
- [47] —, “Xilinx alveo,” <https://www.xilinx.com/products/boards-and-kits/alveo.html>, last accessed 19 May 2022.
- [48] —, “Board support package settings page,” <https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Board-Support-Package-Settings-Page>, last accessed 29 May 2022.
- [49] —, “Dpu on pynq,” <https://github.com/Xilinx/DPU-PYNQ>, last accessed 29 May 2022.
- [50] Dorfell, “Pynq 2.7 for zybo-z7,” <https://discuss.pynq.io/t/pynq-2-7-for-zybo-z7/4124>, last accessed 02 Jun 2022.
- [51] AMD-Xilinx, “Retargeting to a different board,” https://pynq.readthedocs.io/en/latest/pynq_sd_card.html#retargeting-to-a-different-board, last accessed 31 May 2022.
- [52] Logitronix, “Installing tensorflow in pynq,” <https://logitronix.com/wp-content/uploads/2019/04/TensorFlow-Installation-on-PYNQ-Nov-6-2018.pdf>, last accessed 03 Jun 2022.
- [53] K. Hyodo, “Tensorflow-bin previous_versions,” https://github.com/PINTO0309/Tensorflow-bin/tree/main/previous_versions, last accessed 03 Jun 2022.
- [54] Y. Shen, T. Ji, M. Ferdman, and P. Milder, “Argus: An end-to-end framework for accelerating cnns on fpgas,” *IEEE Micro*, vol. 39, no. 5, pp. 17–25, 2019.
- [55] S. Sabogal, A. George, and G. Crum, “Recon: A reconfigurable cnn acceleration framework for hybrid semantic segmentation on hybrid socs for space applications,” in *2019 IEEE Space Computing Conference (SCC)*, 2019, pp. 41–52.
- [56] S. Mouselinos, V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, “Tf2fpga: A framework for projecting and accelerating tensorflow cnns on fpga platforms,” in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4.
- [57] Xilinx, “Zynq ultrascale+ mpsoc,” <https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html>, last accessed 12 Sep 2022.
- [58] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks.” Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA’15, February 2015, pp. 161–170.
- [59] S. I. Venieris and C. S. Bouganis, “Fpgaconvnet: A framework for mapping convolutional neural networks on fpgas.” Proceedings - 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, May 2016, pp. 40–47.
- [60] A. Dunder, J. Jin, B. Martini, and E. Culurciello, “Embedded streaming deep neural networks accelerator with applications,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 7, pp. 1572–1583, July 2017.

- [61] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1–3, pp. 239–255, December 2010.
- [62] N. Li, S. Takaki, Y. Tomioka, and H. Kitazawa, "A multistage dataflow implementation of a deep convolutional neural network based on fpga for high-speed object recognition." 2016 IEEE Southwest Symposium On Image Analysis and Interpretation (SSIAI), 2016, pp. 165–168.
- [63] "Caffe: Deep learning framework," <http://caffe.berkeleyvision.org/>, last accessed 15 Feb 2018.
- [64] "Mathworks: Matlab," <https://www.mathworks.com/products/matlab.html>, last accessed 15 Feb 2018.
- [65] "Microsoft cognitive toolkit," <https://www.microsoft.com/en-us/cognitive-toolkit/>, last accessed 15 Feb 2018.
- [66] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning." Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'14, March 2014, pp. 269–284.
- [67] Y. Zhou and J. Jiang, "An fpga-based accelerator implementation for deep convolutional neural networks." 4th International Conference on Computer Science and Network Technology (ICCSNT), December 2015, pp. 829–832.
- [68] Y. Murakami, "Fpga implementation of a simd-based array processor with torus interconnect." 2015 International Conference on Field Programmable Technology, FPT 2015, May 2015, pp. 244–247.
- [69] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, November 2008.
- [70] B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman, "Systematic literature reviews in software engineering-a tertiary study," *Information and Software Technology*, August 2010.
- [71] A. krizhevsky, "Survey: Implementing dense neural networks in hardware," <https://arxiv.org/abs/1404.5997>, April 2014, last accessed 15 Feb 2018.
- [72] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," <https://arxiv.org/abs/1410.0759>, December 2014, last accessed 15 Feb 2018.
- [73] F. Ortega-Zamorano, J. M. Jerez, D. U. Munoz, R. M. Luque-Baena, and L. Franco, "Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 9, pp. 1840–1850, August 2016.
- [74] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. Lecun, "Neuflow: A runtime reconfigurable dataflow processor for vision." IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, June 2011, pp. 109–116.

- [75] M. R. D. Abdu-Aljabar, "Design and implementation of neural network in fpga," *Journal of Engineering and Development*, vol. 16, no. 3, September 2012.
- [76] G. H. Shakoory, "Fpga implementation of multilayer perceptron for speech recognition," *Journal of Engineering and Development*, vol. 17, no. 6, December 2013.
- [77] E. Z. Mohammed and H. K. Ali, "Hardware implementation of artificial neural network using field programmable gate array," *International Journal of Computer Theory and Engineering*, vol. 5, no. 5, October 2013.
- [78] S. Singh, S. Sanjeevi, S. V., and A. Talashi, "Fpga implementation of a trained neural network," *IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)*, vol. 10, no. 3, May-June 2015.
- [79] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor." Proceedings of the 42nd Annual International Symposium on Computer Architecture-ISCA'15, June 2015, pp. 92–104.
- [80] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks." 21st Asia and South Pacific Design Automation Conference, 2016, pp. 575–580.
- [81] L. B. Saldanha and C. Bobda, "Sparsely connected neural networks in fpga for handwritten digit recognition." Proceedings - International Symposium on Quality Electronic Design (ISQED), May 2016, pp. 113–117.
- [82] Y. Wang, L. Xia, T. Tang, B. Li, S. Yao, M. Cheng, and H. Yang, "Low power convolutional neural networks on a chip," no. 1. IEEE International Symposium on Computer Architecture, April 2016, pp. 129–132.
- [83] C. Kyrkou, C. S. Bouganis, T. Theocharides, and M. M. Polycarpou, "Embedded hardware-efficient real-time classification with cascade support vector machines," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 1, January 2016.
- [84] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen, "Dadiannao: A neural network supercomputer," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 73–88, January 2017.
- [85] "Face image analysis with convolutional neural networks," https://lmb.informatik.uni-freiburg.de/papers/download/du_diss.pdf, last accessed 15 Feb 2018.
- [86] Z. Saidane, *Image and video text recognition using convolutional neural networks: Study of new CNNs architectures for binarization, segmentation and recognition of text images*. LAP LAMBERT Academic Publishing, 2011.

State of the art of the Hardware Neural Networks (HNNs)

With the appearance of high performance platforms in the last decade (i.e. GPU, FPGAs), Neural Networks (NNs) are becoming an attractive tool for many classification and object recognition problems. Despite this, the existent APIs for running NNs don't exploit the hardware resources efficiently and for this reason several researches have turn back their attention to Hardware Neural Networks (HNNs) [58], [59], [60]. HNNs are the implementation of accelerators architectures that evaluate the NNs forward propagation algorithms, by using reconfigurable platforms like FPGAs, or hardcore designs like the VLSI circuits [61], [62]. Usually, the networks are trained by means of tools such as Caffe [63], MATLAB [64], TensorFlow [19], Microsoft Cognitive Toolkit [65], etc. and then, with the weights and bias calculated, a hardware accelerator is implemented [66], [67], [68]. Nevertheless, the discussion of how to design HNNs had been widen due to number of works that had been proposed in the last few years [61], and up to day, a new literature review is needed to identify the most relevant search streams that had appeared lately, the NN types being used, the organizations leading the research, the research limitations and the quality of the implementation frameworks being proposed. In this chapter, we aim to answer all of these questions by means of a Systematic Literature Review (SLR) [69], [70]. This chapter is organized as follows; Section I describes the carried out Systematic Literature Review. Section II presents the SLR results and Section III the discussion. Finally, the conclusion is drawn in Section IV and the Research Questions and Hypothesis in sections V and VI.

1 Systematic Literature Review Method

The literature review was made by applying the Systematic Literature Review (SLR) method, proposed by Kitchenham et al. in [69] and [70]. The goal of this review is to identify the gaps existing in the HNN implementation process by studying the relevant works in the state of the art. Additionally, according to the literature NN implementations on traditional platforms like general purpose processors don't use hardware resources efficiently, and therefore there is a growing interest in exploring others platforms [58], [59], [60]. On the other hand, Graphical Processing Units (GPUs) are being widely used for training NN because of their high throughput, but those implementations are limited by the local memory available and the communication

bandwidth between the host and the GPU, [71], [72], [60] besides, its power consumption make them not practical for embedded systems. Fortunately, these implementations can be improved by using accelerators in custom hardware (e.g. ASICs, FPGA), which have demonstrated to have better performance during the feedforward prediction stage and low power consumption [60], [73]. Moreover, FPGAs are easily programmable and inexpensive, these being why they are the most advantageous option. Furthermore, with the appearance of more optimization design techniques and the amount of FPGA logic resources available being increased, the design exploration space is enlarged boosting the HNN implementation on FPGA-based platforms. By taking into account the above-mentioned factors, it was decided to focus this literature review on HNN works and implementation frameworks designed for these platforms, without including GPU-based works. Lastly, the SLR method steps are documented below.

1.1 Research Questions (RQ)

The research questions addressed by this review are:

- RQ1: How many frameworks for implementing HNN has been proposed since 2010?
- RQ2: What types of Neural Networks are being addressed?
- RQ3: What individuals and organizations are leading the research?
- RQ4: What are the limitations of current research?
- RQ5: Is the quality of the implementation frameworks improving?

With respect to RQ1, the review starts at 2010 because HNNs start to be a feasible option when the resources and computation platforms available were sufficient for the efficient implementation around 2010. With respect to RQ2, there are several types of NN that can be implemented in hardware, however, differences in input data, classification tasks, activation function, etc. can lead to choose one instead of another. For this reason it is important to know which are the NN types being used for HNN. With respect to RQ3, it is essential to identify HNN research trends, relevant authors and leading works to be aware of the current problems being studied. With respect to limitations of HNN research (RQ4) the following issues are going to be considered:

- RQ4.1: Were the scope of HNN implementation frameworks limited?
- RQ4.2: Is there evidence that the use of HNNs is limited due to lack of implementation frameworks?
- RQ4.3: Is the quality of implementation frameworks appropriate?
- RQ4.4: Are frameworks contributing to the implementation of HNN by defining practice guidelines?

With respect to RQ5, it is important to know if the proposed frameworks are being improved in subsequent works or if the new proposed frameworks are taking different approaches.

1.2 Search Process

The IEEE Computer Society Digital Library, the SCOPUS indexing system and Open Access organizations like IAES, IOSR were used in the search process. All searches were based on titles, keywords and abstracts of works published in journals, conferences and symposiums since 2010, which are shown in Table A.1. The

Source	Acronym	Organization	Publication
Transactions on Neural Networks and Learning Systems	TNNLS	IEEE	Journal
Transactions on Very Large Scale Integration Systems	TVLSIS	IEEE	Journal
Transactions on Computers Neural Networks	TC NN	IEEE ELSEVIER	Journal Journal
Neurocomputing	NC	ELSEVIER	Journal
Information Fusion	IF	ELSEVIER	Journal
Computer Methods and Programs in Biomedicine	CMPB	ELSEVIER	Journal
Pattern Recognition	PR	ELSEVIER	Journal
Engineering Applications of Artificial Intelligence	EAAI	ELSEVIER	Journal
Engineering Research And Development	IJERD	Peer Reviewed	Journal
Electrical and Computer Engineering	IJECE	IAES	Journal
Electronics and Communication Engineering	JECE	IOSR	Journal
International Conference on Data Mining Workshops	ICDMW	IEEE	Conference
International Conference on Computer Science and Network Technology	ICCSNT	IEEE	Conference
International Conference on Architectural Support for Programming Languages and Operating Systems	ASPLOS	IEEE/ACM	Conference
International Conference on Parallel Architectures and Compilation Techniques	PACT	IEEE/ACM	Conference
Annual International Symposium on Computer Architecture	ISCA	IEEE/ACM	Symposium
Annual International Symposium on Field Programmable Custom Computing Machines	ISFPCCM	IEEE	Symposium
Annual International Symposium on Field Programmable Gate Arrays	FPGA	IEEE/ACM	Symposium

Table A.1: Selected journals, symposiums and conference proceedings.

search string used in the IEEE library was “Neural Networks” AND “Hardware” and the search string used in SCOPUS was TITLE-ABS-KEY(“Neural Networks”) AND TITLE-ABS-KEY(“Hardware”) OR TITLE-ABS-KEY(“Framework”).

1.3 Study selection

The results for the different searches were added, obtaining a total number of 491 papers published between Jan 1st 2010 and March 31th 2017: 143 from the IEEE digital library, 343 from SCOPUS indexing system and 5 from the open access organizations. To these papers, the following inclusion and exclusion criteria were applied.

Topics used to include the papers:

- Frameworks for implementing HNN with defined research questions, search process, data extraction and data presentation, whether or not the researchers referred to their study as a implementation framework.
- Approach to optimize current implementations of HNN.

Papers on the following topics were excluded:

- Informal literature surveys (no defined research questions; no defined search process; no defined data extraction process).
- Papers presenting implementations of HNN with not defined procedures or not discussing the procedures used.
- Papers presenting GPU-based works, which are limited by local memory constraints and bandwidth communication will also be excluded.
- Duplicate reports of the same study (when several reports of a study exist in different journals the most complete version of the study was included in the review).

After excluding papers that were obviously irrelevant, had not enough information, or were duplicates, there were 61 papers remaining. Those papers were then subject to a more detailed assessment, where each paper was reviewed to identify papers that could be rejected on the basis that they did not include literature reviews, or that they were not related to implementation frameworks. This led to the exclusion of 41 papers. The remaining papers are shown in Table A.2.

1.4 Quality assessment (QA)

Each article was evaluated using the following quality assessment (QA) questions based on [69]:

- QA1: Is the HNN implementation process presented explicitly?
- QA2: Is the literature search likely to have covered all relevant studies?
- QA3: Did the HNN implementation assess the quality/validity of previous included studies?
- QA4: Were the basic data/studies adequately described?

There are three possible outputs for each question with the following score: Y (*yes*) = 1, P (*partly*) = 0.5 and N (*no*) = 0. For QA1: Y the implementation process is presented explicitly, P the implementation process is implicit and N the implementation process is not defined and cannot be readily inferred.

For QA2: Y the authors had cited at least 20 works including highly cited works, P the authors had cited between 15 and 19 works including relevant works. N the authors had cited less than 15 works or they had cited irrelevant works.

For QA3: Y The HNN implementation performance improved former ones in more than 2x, P the performance was less than 2x of previous ones, and N performance was not reported.

For QA4: Y information of each primary study is presented, P each primary study is barely presented, and N any information of each primary study is given.

1.5 Data collection

The data extracted from each work were:

- The source (journal, conference or symposium) and full reference.

- Classification of the study type (i.e. HNN implementation, VLSI design, HNN implementation frameworks).
- Main topic area.
- The author(s), their institution and the country where it is situated.
- Summary of the study including the main research questions and the answers.
- Research question/issue.
- Quality evaluation.
- How many primary studies were used in the work.

1.6 Data analysis

The data was tabulated (see Tables A.1 and A.3) to show:

- The number of HNN works published per year and their source (addressing RQ1).
- Whether the HNN work referenced others papers (addressing RQ1).
- The topics studied by the HNN works, i.e. HNN implementation, VLSI design, HNN implementation frameworks (addressing RQ2 and RQ4.1).
- The authors: the affiliations of the authors and their institutions was reviewed but not tabulated (addressing RQ3).
- The number of previous HNN works in each paper (addressing RQ4.2).
- The quality score for each HNN work (addressing RQ4.3).

2 Results

2.1 Search Results

Including works from journals, symposiums and conferences they were 20 papers reviewed. These papers are shown in Table A.2.

2.2 Quality evaluation of the HNN works.

The HNN works shown in Table A.2 were assessed based on the quality assessment (QA) questions. These results are shown in Table A.3.

2.3 Quality factors

The average Quality Scores (QS) for studies each year, the mean and the standard deviation σ are shown in Table A.4. As can be seen the number of HNN studies in the last few years has grew up from 1 study per year up to 9 studies, showing the growing interest for HNN. Also, the average QS per year has been quasi-stable around 3.0 (i.e. 2.88), which can be seen as an increase in the number of most comprehensive works on the topic.

Study Ref.	Authors	Date	Paper type	Number primary studies	Review topics
[61]	Misra & Saha	2010	Journal	278	Overview of HNN models: HNN chips, Cellular HNN, Neuromorphic HNN, Optical NN.
[74]	Farabet et al.	2011	Conference	27	HNN implementation, hardware architectures.
[75]	Rana D. Abdu-Aljabar	2012	Journal	23	HNN implementation.
[76]	Shakoory	2013	Journal	12	HNN implementation.
[77]	Mohammed et al.	2013	Journal	9	HNN implementation.
[66]	Chen et al.	2014	Conference	44	HNN implementation, VLSI design.
[78]	Singh et al.	2015	Journal	11	HNN implementation.
[58]	Zhang et al.	2015	Symposium	16	HNN implementation.
[67]	Zhou, Y., & Jiang, J.	2015	Conference	12	HNN implementation.
[79]	Du et al.	2015	Symposium	61	HNN implementation, VLSI design.
[59]	Venieris et al.	2016	Symposium	16	HNN implementation framework.
[68]	Murakami, Y.	2016	Conference	8	HNN implementation.
[80]	Motamedi et al.	2016	Conference	10	HNN Parallelism.
[60]	Dundar et al.	2016	Journal	49	HNN implementation.
[62]	Li et al.	2016	Symposium	11	HNN implementation.
[81]	Saldanha et al.	2016	Symposium	13	HNN implementation.
[82]	Wang et al.	2016	Symposium	19	HNN implementation, VLSI design.
[73]	Ortega-Zamorano et al.	2016	Journal	44	HNN implementation framework for Backpropagation.
[83]	Kyrkou et al.	2016	Journal	41	HNN implementation
[84]	Luo et al.	2017	Journal	66	HNN implementation, VLSI design.

Table A.2: Systematic Review of HNN Studies.

3 Discussion

The answers to the research questions are discussed in this section.

- RQ1: How many frameworks for implementing HNN has been proposed since 2010?

The revision of several studies from 2010 to 2017 lead to 20 relevant HNN studies. Moreover, it is observed that the interest in HNN is growing up as the number of studies per year.

Relevant studies included 1 survey of the HNN implementations proposed before 2010 [61], 4 studies of HNN Very Large Scale Integration (VLSI) designs [66], [79], [84], [82], and the rest of studies proposed an HNN design implemented in a reconfigurable platform (e.g. FPGA). Despite not all of them presented an explicit framework, the implementation process could be readily inferred.

- RQ2: What types of Neural Networks are being addressed?

Most of the works aimed to implement Convolutional Neural Networks (CNN) [85], [86]. CNN has been highly accepted in classification and object recognition problems because of its accuracy and relatively

Study Ref.	Paper type	QA ₁	QA ₂	QA ₃	QA ₄	Total Score
[61]	Journal	N	Y	N	Y	2.0
[74]	Conference	Y	Y	Y	P	3.5
[75]	Journal	Y	P	P	Y	3.0
[76]	Journal	Y	N	P	P	2.0
[77]	Journal	N	N	N	N	0.0
[66]	Conference	Y	Y	Y	Y	4.0
[78]	Journal	Y	N	P	N	1.5
[58]	Symposium	Y	P	Y	P	3.0
[67]	Conference	P	P	P	P	2.0
[79]	Symposium	Y	Y	Y	Y	4.0
[59]	Symposium	Y	P	Y	Y	3.5
[68]	Conference	Y	N	P	P	2.0
[80]	Conference	Y	P	Y	Y	3.5
[60]	Journal	Y	Y	Y	Y	4.0
[62]	Symposium	Y	P	P	N	2.0
[81]	Symposium	P	P	P	N	1.5
[82]	Symposium	P	P	P	P	2.0
[73]	Journal	Y	Y	Y	Y	4.0
[83]	Journal	Y	Y	Y	Y	4.0
[84]	Journal	Y	Y	Y	Y	4.0

Table A.3: Quality evaluation of the HNN studies.

		Year							
		2010	2011	2012	2013	2014	2015	2016	2017
#	of	1	1	1	2	1	4	9	1
	papers								
QS		2.0	3.5	3.0	1.0	4.0	2.625	2.94	4
Mean									
QS σ		0.88	0.625	0.12	1.88	1.12	0.255	0.06	1.12

Table A.4: Average Quality Scores (QS) for studies by publication date.

fair cost. CNNs are a class of Deep Neural Networks (DNN) where weights are shared across neurons, thus reducing the memory needed to stored the training parameters.

- RQ3: What individuals and organizations are leading the research?

The leadership of HNNs implementation can be divided by approach. The VLSI design of HNNs is lead by the work group form by the State Key Laboratory of Computer Architecture in China, the Institute of Computing Technology processing in China and the Inria Institution in France. They had designed and implemented at least 4 different HNN chips [66], [79], [82].

On the other hand, there are different authors that had contributed with several studies about HNNs implementation in reconfigurable platforms. For example, Eugenio Culurciello from the Courant Institute of Mathematical Sciences, New York University and Yann LeCun from the Electrical Engineering Department, Yale University both in USA presented works that include a dataflow processor for vision [74],

and an Embedded Streaming DNN Accelerator [60]. In addition, Stylianos Veneris and Christos-Savvas Bougaris from the Department of Electrical and Electronic Engineering, Imperial College in London had presented fpgaConvNet, a framework for mapping CNN on FPGAs in [59]. Finally, Francisco Ortega-Zambrano et al. from the Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga in Spain had proposed an efficient implementation of the NN training stage on FPGA in [73].

- RQ4: What are the limitations of current research?

Currently, due to the number of computational resources (i.e. memory and processing) needed by HNNs implementations, researches aim to commercial FPGAs-based systems (i.e. mostly large FPGAs), while the design of HNNs VLSI chips remains limited. Moreover, there are other factors that restrict the VLSI research such as: the power consumption, die size, fabrication technology and cost. Another important reason is that there is not a general agreement between approaches and there are studies that consider parameters that are not important to others studies, widening the exploration space but reducing the concentrated efforts. Thus, there are a few implementation frameworks proposed and their quality varies from barely acceptable to regular.

- RQ5: Is the quality of the implementation frameworks improving?

Due to the growing interest in implementing HNN, the number of proposed frameworks by year has been increasing, and so the quality of studies published. For example, current studies offer a more complete description of the proposed work, make comparisons with similar studies and provide external links that widen the information available.

4 Conclusions

This chapter presented a Systematic Literature Review of the latest works related to the implementation of neural networks into hardware, from which two main streams can be identified: HNNs VLSI designs and HNNs implementation in reconfigurable platforms. Hence, the last one has the majority of studies due to the known constrictions of the VLSI fabrication process. Also, it was found that CNNs are the most aimed NN because of its features and applications. In addition, the awakened interest in HNNs has revealed the necessity of implementation frameworks, that allow to identify relevant parameters and that present a solid number of stages for accomplish the implementation. Unfortunately, frameworks available in the state of the art lack of simplicity, usually aim to bigger hardware platforms, have an excessive use of logic resources and present an acceptable accuracy.

Moreover, there are several problems in the implementation process that still had to be tackle down like: memory bottlenecks, scarce number of resources, complexity of the NNs, implementation precision and accuracy, and efficient HNNs training.

5 Research Questions

These are the research questions that emerge from the literature review.

- How to create a framework that allow to implement neural networks in hardware?
- Is it possible to implement a NN accelerator to approach different problems?

- What are the NN parameters that affect the implementation of NN in hardware platforms?

6 Hypothesis

This section presents some hypothesis formulated around the research questions:

- How the input data type and the processing tasks can be used to determine the NN topology and the design of the HNN accelerator?
- How the different works that can be found in the literature can be use to implement a cost-effective and easy-deployment framework that aim to solve different problems?
- How are the number of neurons per layer related with the number of computational resources needed to implement the HNN in a hardware platform?
- How are the neurons activation function related with the amount of computational resources needed to implement the HNN in a hardware platform?
- How the use of fixed point or floating point algorithms affects the NN accuracy?

Useful Scripts

Listing B.1: 00_M6_jaffe_6emo.py

```
1 import matplotlib as mpl
2 mpl.rcParams['legend.fontsize'] = 12;
3 mpl.rcParams['axes.labelsize'] = 12;
4 mpl.rcParams['xtick.labelsize'] = 12;
5 mpl.rcParams['ytick.labelsize'] = 12;
6 mpl.rcParams['text.usetex'] = True;
7 mpl.rcParams['font.family'] = 'sans-serif';
8 mpl.rcParams['mathtext.fontset'] = 'dejavusans';
9 mpl.rcParams.update({'font.size': 12});
10 import matplotlib.pyplot as plt
11
12 import numpy as np
13 import os
14 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'; # TF debug messages
15 # 0 = all messages are logged (default behavior)
16 # 1 = INFO messages are not printed
17 # 2 = INFO and WARNING messages are not printed
18 # 3 = INFO, WARNING, and ERROR messages are not printed
19
20 import sys
21 np.set_printoptions(threshold=sys.maxsize) # Printing all the weights
22 import tempfile
23 import tensorflow as tf
24 physical_devices = tf.config.list_physical_devices('GPU')
25 tf.config.experimental.set_memory_growth(physical_devices[0], True)
26 from tensorflow import keras
```

```

27
28 from matplotlib.backends.backend_pdf import PdfPages
29
30
31 print(" \n ");
32 print("*****");
33 print(" Charger l'ensemble de donnÃ©es ");
34 print("*****");
35 # Expressions: angry, disgust, fear, happy, sad, surprise, neutral
36 fer_eti = ["HA", "AN", "DI", "FE", "SA", "SU"]; # Expressions abrÃ©gÃ©es
37
38 fer_lab = [ 0, 1, 2, 3, 4, 5 ]; # Index des expressions
39
40 nom_cls = 6; # Nombre des classes.
41
42 # 10-fold cross validation: La totalitÃ© de l'ensemble de donnÃ©es est
43 # divisÃ© en 10 sets, 9 pour l'entraÃªnement et 1 pour le testing.
44
45 # Charger entraÃªnement: Jaffe + prÃ©-traitement avec dlib + lbp-var + 15
46 # augmentations
47 x_train = np.load("../db_transfigure/jaffe_dlib_var64_aug15_6emo_img.npy");
48 y_train = np.load("../db_transfigure/jaffe_dlib_var64_aug15_6emo_lab.npy");
49
50 # Charger testing: Jaffe + prÃ©-traitement avec dlib + lbp-var
51 x_test = np.load("../db_transfigure/jaffe_dlib_var64_test_6emo_img.npy");
52 y_test = np.load("../db_transfigure/jaffe_dlib_var64_test_6emo_lab.npy");
53
54 print("Ensemble des donnÃ©es: jaffe_dlib_var64_aug15_6emo_img.npy");
55 #print("Img dat: ", img_dat, img_dat.shape, type(img_dat), img_lab);
56 print("Taille des images:", x_train.shape, type(x_train));
57 print("Taille des Ã©tiquettes: ", y_train.shape, type(y_train));
58
59 x_train = np.array(x_train, "float32");
60 x_test = np.array(x_test, "float32");
61 print(x_train.shape[0], 'train samples');
62 print(x_test.shape[0], 'test samples');
63
64 # Dessiner quelques donnÃ©es
65 #fig = plt.figure();

```

```

63 #ax1 = fig.add_subplot(2, 2, 1); ax1.set_title(fer_eti[y_train[0]]); ax1.imshow(
    x_train[0], cmap="gray");
64 #ax2 = fig.add_subplot(2, 2, 2); ax2.set_title(fer_eti[y_train[1]]); ax2.imshow(
    x_train[1], cmap="gray");
65 #ax3 = fig.add_subplot(2, 2, 3); ax3.set_title(fer_eti[y_train[2]]); ax3.imshow(
    x_train[2], cmap="gray");
66 #ax4 = fig.add_subplot(2, 2, 4); ax4.set_title(fer_eti[y_train[3]]); ax4.imshow(
    x_train[3], cmap="gray");
67 #fig.suptitle("Images avec prÃ©-traitement", fontsize=12); plt.tight_layout();
68 #fig.subplots_adjust(top=0.88);
69 #plt.show();
70 #sys.exit(0); # Terminer l'execution
71
72
73 print ( " \n " );
74 print ( "*****" );
75 print ( " ModÃ©le " );
76 print ( "*****" );
77 # DÃ©finir l'architecture du modÃ©le
78 model = keras.Sequential( [
79     keras.layers.InputLayer( input_shape=(64, 64) ),
80     keras.layers.Reshape( target_shape=(64, 64, 1) ),
81
82     keras.layers.Conv2D( filters=32, kernel_size=(5, 5), strides=(1,1), padding="
        SAME", activation="relu"),
83     keras.layers.MaxPooling2D( pool_size=(2, 2), strides=(2,2) ),
84
85     keras.layers.Conv2D( filters=64, kernel_size=(5, 5), strides=(1,1), padding="
        SAME", activation="relu"),
86     keras.layers.MaxPooling2D( pool_size=(2, 2), strides=(2,2) ),
87
88     keras.layers.Conv2D( filters=128, kernel_size=(5, 5), strides=(1,1), padding="
        SAME", activation="relu"),
89     keras.layers.MaxPooling2D( pool_size=(2, 2), strides=(2,2) ),
90
91     keras.layers.Flatten( ),
92     keras.layers.Dense(nom_cls, activation=None) ] );
93
94 model.summary();
95 keras.utils.plot_model(model, to_file="model.png", show_shapes="True");

```

```

96 #sys.exit(0); # Terminer l'execution
97
98
99 print ("***** ");
100 print (" Entraînement... ");
101 print ("***** ");
102 # Paramètres d'entraînement
103 batch_size = 32; # Mini-batch size.
104 epochs = 30; # Nombre des époques.
105 lr_val = 0.001; # Learning rate.
106 print ("batch_size = ", batch_size);
107 print ("learning rate = ", lr_val);
108
109 # Entraînement ...
110 opt = keras.optimizers.Adam(learning_rate=lr_val);
111 model.compile(optimizer=opt,
112               loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
113               metrics=['accuracy']);
114
115 his = model.fit( # Sauvegarder l'
116                 histoire
117                 x_train, y_train, batch_size=batch_size,
118                 steps_per_epoch = len(x_train) / batch_size,
119                 epochs=epochs, verbose=2,
120                 validation_data=(x_test, y_test), workers=8);
121 #sys.exit(0); # Terminer l'execution
122
123 print ("***** ");
124 print (" Evaluation de performance ");
125 print ("***** ");
126 # Evaluating
127 score = model.evaluate(x_test, y_test, verbose=0);
128 print ('Avec la collection de test');
129 print ('--> Perte:', score[0]);
130 print ('--> Exactitude:', 100*score[1]);
131
132 predictions = model.predict(x_test);
133 predicted_label = np.argmax(predictions[0]);
134

```

```

135 #print(history.history);
136 #print(his.history.keys());
137
138 # Fonction qui sert à mesurer le performance du modèle
139 with PdfPages("graphique_entrainement.pdf") as export_pdf:
140
141     # summarize history for accuracy
142     plt.plot(his.history['accuracy']);
143     plt.plot(his.history['val_accuracy']);
144     plt.title('Exactitude du modèle');
145     plt.ylabel('Exactitude'); plt.xlabel('Époque');
146     plt.legend(['train', 'test'], loc='upper left');
147     export_pdf.savefig(); plt.close();
148
149     # summarize history for loss
150     plt.plot(his.history['loss']);
151     plt.plot(his.history['val_loss']);
152     plt.title('Perte du modèle');
153     plt.ylabel('Perte'); plt.xlabel('Époque');
154     plt.legend(['train', 'test'], loc='upper left');
155     export_pdf.savefig(); plt.close();
156
157
158 # En sauvegardant le modèle en hdf5
159 #-----
160 model.save("models/M6_6emo.h5");
161 model.save_weights("models/M6_6mo_weights.h5");
162 #sys.exit(0);
163
164 print("*****");
165 print("*   Merci d'utiliser ce logiciel!   *");
166 print("*           (8-)           *");
167 print("*****");

```

Training M6 for FER with PyTorch

Entrée [33]: %matplotlib inline

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import torchvision
import torchvision.transforms as transforms

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

Entrée [34]: from torch.utils.data import TensorDataset, DataLoader

```
# Charger entraînement: Jaffe + pré-traitement avec dlib + lbp-var + 15 augmentations
x_train = np.load("db_transfigure/jaffe_dlib_var64_aug15_6emo_img.npy");
y_train = np.load("db_transfigure/jaffe_dlib_var64_aug15_6emo_lab.npy");
x_train = x_train[500:1000][:][:]; y_train = y_train[500:1000][:][:];

# Charger testing: Jaffe + pré-traitement avec dlib + lbp-var
x_test = np.load("db_transfigure/jaffe_dlib_var64_test_6emo_img.npy");
y_test = np.load("db_transfigure/jaffe_dlib_var64_test_6emo_lab.npy");

print("Ensemble des données: jaffe_dlib_var64_aug15_6emo_img.npy");
print("Img dat: ", x_train, x_train.shape, type(x_train), x_train[0]);
print("Taille des images: ", x_train.shape, type(x_train));
print("Taille des étiquettes: ", y_train.shape, type(y_train));

# Convertir à PyTorch
x_train = torch.from_numpy(x_train);
y_train = torch.from_numpy(y_train);
x_test = torch.from_numpy(x_test);
y_test = torch.from_numpy(y_test);

# Create loaders à partir du tenseurs en *.npy
trainset = TensorDataset(x_train, y_train);
```

1 sur 6

M6_jaffe_6emo_pytorch - Jupyter Notebook

22/07/2022, 17:37

http://localhost:8888/notebooks/M6_jaffe_6emo_pytorch.ipynb

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=1,
                                          shuffle=True, num_workers=2);
testset = TensorDataset(x_test, y_test);
testloader = torch.utils.data.DataLoader(testset, batch_size=1,
                                         shuffle=True, num_workers=2);

# Expressions: angry, disgust, fear, happy, sad, surprise, neutral
fer_eti = ["HA", "AN", "DI", "FE", "SA", "SU"]; # Expressions abrégés.
fer_lab = [ 0, 1, 2, 3, 4, 5 ]; # Index des expressions.
```

Ensemble des données: jaffe_dlib_var64_aug15_6emo_img.npy

```
Img dat: [[ [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 4 3 2]
 [ 0 0 0 0 ... 0 0 0]
 ...
 [ 0 1 0 0 ... 0 1 0]
 [ 0 1 1 0 ... 1 1 0]
 [ 0 0 0 0 ... 2 2 0]]

[[ [ 3 3 3 3 ... 0 0 0]
 [ 4 3 5 5 ... 0 0 0]
 [ 6 4 3 3 ... 0 0 0]
 ...
 [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]]

[[ [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 3 0 0]
 ...
 [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]]

...

[[ [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]
 [ 0 0 0 0 ... 0 0 0]
 ...
 [ 0 0 0 0 ... 18 27 33]
 [ 0 0 0 0 ... 44 31 15]
 [ 0 0 0 0 ... 0 0 0]]
```

2 sur 6

22/07/2022, 17:37

Figure B.1: Notebook for M6 model in PyTorch installed on Pynq 1.8 running on the Zybo-Z7 board.


```

[[ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 ...
 [ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]]

[[ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 ...
 [ 2  2  3 ...  0  0  0]
 [ 1  2  2 ...  0  0  0]
 [ 1  1  2 ...  0  0  0]]] (500, 64, 64) <class 'numpy.ndarray'> [[0 0 0 ... 0 0 0]
[0 0 0 ... 4 3 2]
[0 0 0 ... 0 0 0]]
...
[0 1 0 ... 0 1 0]
[0 1 1 ... 1 1 0]
[0 0 0 ... 2 2 0]]
Taille des images: (500, 64, 64) <class 'numpy.ndarray'>
Taille des étiquettes: (500,) <class 'numpy.ndarray'>

```

3 sur 6

M6_jaffe_6emo_pytorch - Jupyter Notebook

22/07/2022, 17:37

http://localhost:8888/notebooks/M6_jaffe_6emo_pytorch.ipynb

```

Entrée [35]: import matplotlib.pyplot as plt
import numpy as np

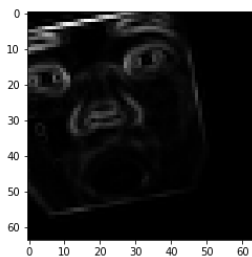
# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next();

# show images
npimg = images[0].numpy();
plt.imshow(npimg, cmap="gray");
print(type(images[0]), images.shape);

# print labels
print(fer_eti[labels[0]]);

<class 'torch.Tensor'> torch.Size([1, 64, 64])
SU

```



```

Entrée [36]: class M6FER_Net(nn.Module):
def __init__(self):
# Here all the layers are defined
super(M6FER_Net, self).__init__()
# Conv2d(#channels(rgb), #filters, kernel_size)
self.conv1 = nn.Conv2d(1, 32, 5);
self.conv2 = nn.Conv2d(32, 64, 5);
self.conv3 = nn.Conv2d(64, 128, 5);
self.mpool = nn.MaxPool2d(2, 2);
self.fc1 = nn.Linear(128 * 4 * 4, 6);

```

4 sur 6

22/07/2022, 17:37

Figure B.2: Notebook for M6 model in PyTorch installed on Pynq 1.8 running on the Zybo-Z7 board.

```

def forward(self, x):
    # Here the model (network) structure is defined
    x = self.mpool(F.relu(self.conv1(x)));
    x = self.mpool(F.relu(self.conv2(x)));
    x = self.mpool(F.relu(self.conv3(x)));
    x = x.view(-1, 128 * 4 * 4)
    x = self.fc1(x);
    return x

net = M6FER_Net()

```

Entrée [37]: `criterion = nn.CrossEntropyLoss(); # sparse categorical cross-entropy (i.e. takes integers as targets instead of floats)`
`optimizer = optim.Adam(net.parameters(), lr=0.001);`

Entrée [38]: `for epoch in range(20): # loop over the dataset multiple times`

```

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data;

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        #outputs = net(inputs.float());
        outputs = net(inputs.view(1, inputs.shape[0], inputs.shape[1], inputs.shape[2]).float());

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 500 == 499: # print every 500 mini-batches
            print('[%d, %5d] Loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 500))
            running_loss = 0.0

print('Finished Training')

```

5 sur 6

M6_jaffe_6emo_pytorch - Jupyter Notebook

22/07/2022, 17:37

http://localhost:8888/notebooks/M6_jaffe_6emo_pytorch.ipynb

```

[1, 500] loss: 1.932
[2, 500] loss: 1.788
[3, 500] loss: 1.759
[4, 500] loss: 1.732
[5, 500] loss: 1.821
[6, 500] loss: 1.826
[7, 500] loss: 1.591
[8, 500] loss: 1.373
[9, 500] loss: 1.194
[10, 500] loss: 0.977
[11, 500] loss: 0.841
[12, 500] loss: 0.904
[13, 500] loss: 0.680
[14, 500] loss: 0.529
[15, 500] loss: 0.482
[16, 500] loss: 0.397
[17, 500] loss: 0.426
[18, 500] loss: 0.284
[19, 500] loss: 0.270

```

Entrée [39]: `correct = 0`
`total = 0`
`with torch.no_grad():`
`for data in testloader:`
 `images, labels = data`
 `#outputs = net(images.float());`
 `outputs = net(images.view(1, images.shape[0], images.shape[1], images.shape[2]).float());`
 `_, predicted = torch.max(outputs.data, 1)`
 `total += labels.size(0)`
 `correct += (predicted == labels).sum().item()`
`print('Accuracy of the network on the 18 test images: %d %%' % (`
 `100 * correct / total))`

Accuracy of the network on the 18 test images: 55 %

6 sur 6

22/07/2022, 17:37

Figure B.3: Notebook for M6 model in PyTorch installed on Pynq 1.8 running on the Zybo-Z7 board.

Complementary Resources

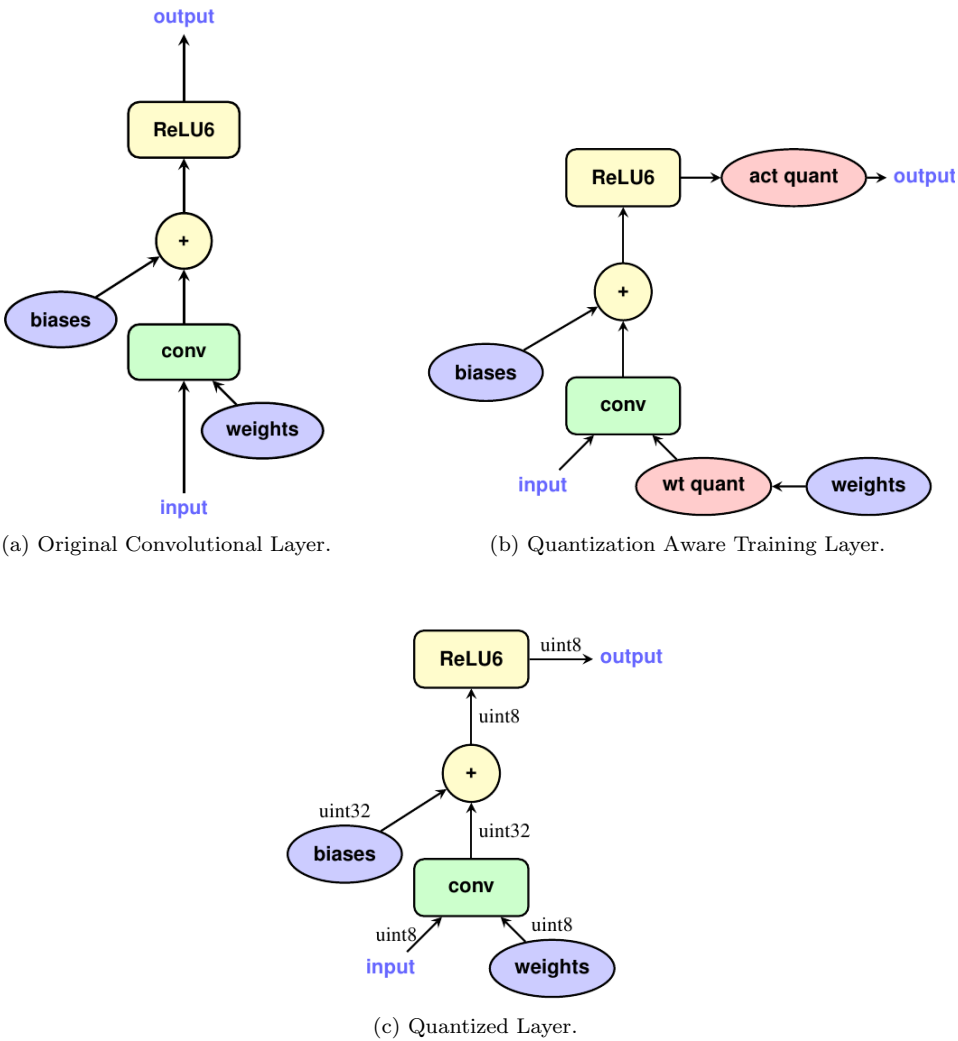


Figure C.1: Tflite Quantization Aware Training. Taken from [13].

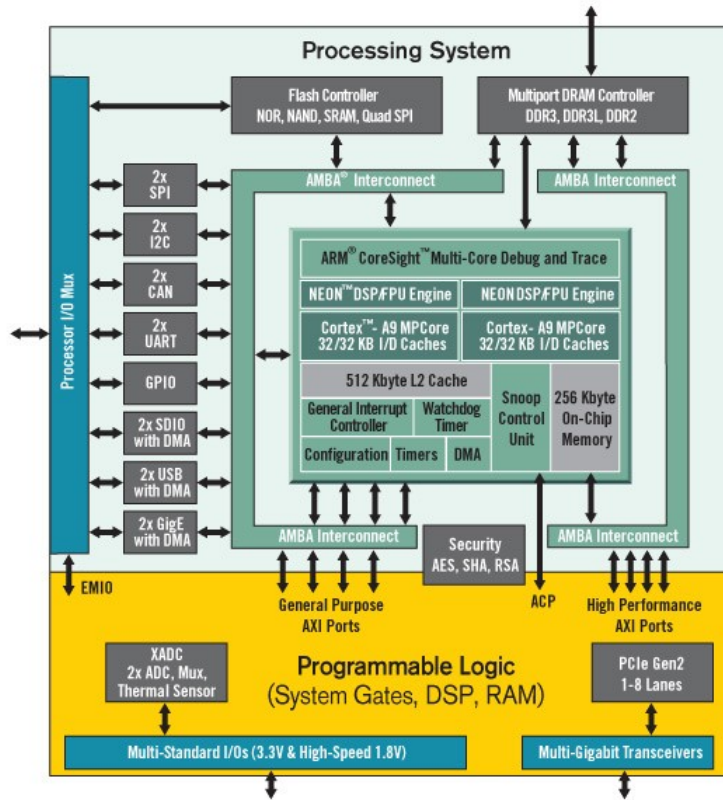
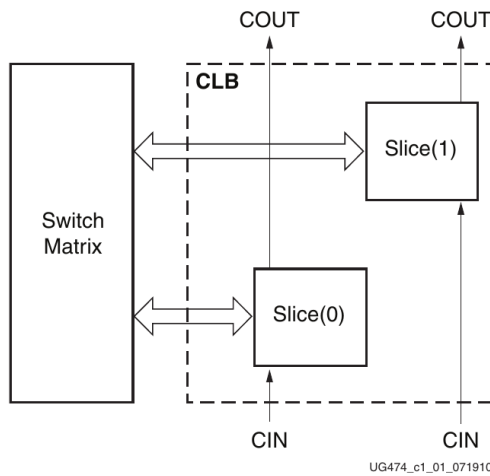


Figure C.2: Zynq APSoC Architecture. Taken from [8].



(a) CLB structure.

Figure C.3: PL architecture. Taken from [9].

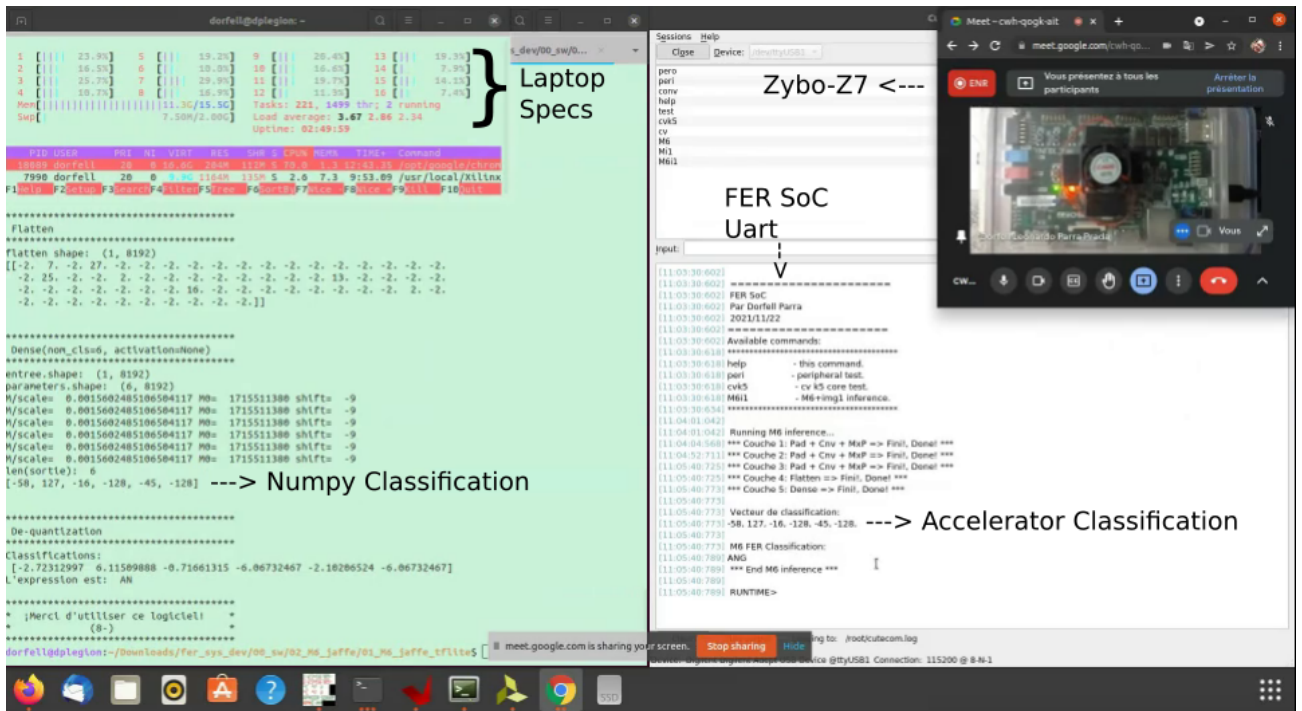


Figure C.4: Validation of results. Comparison between the Numpy and the SoC.

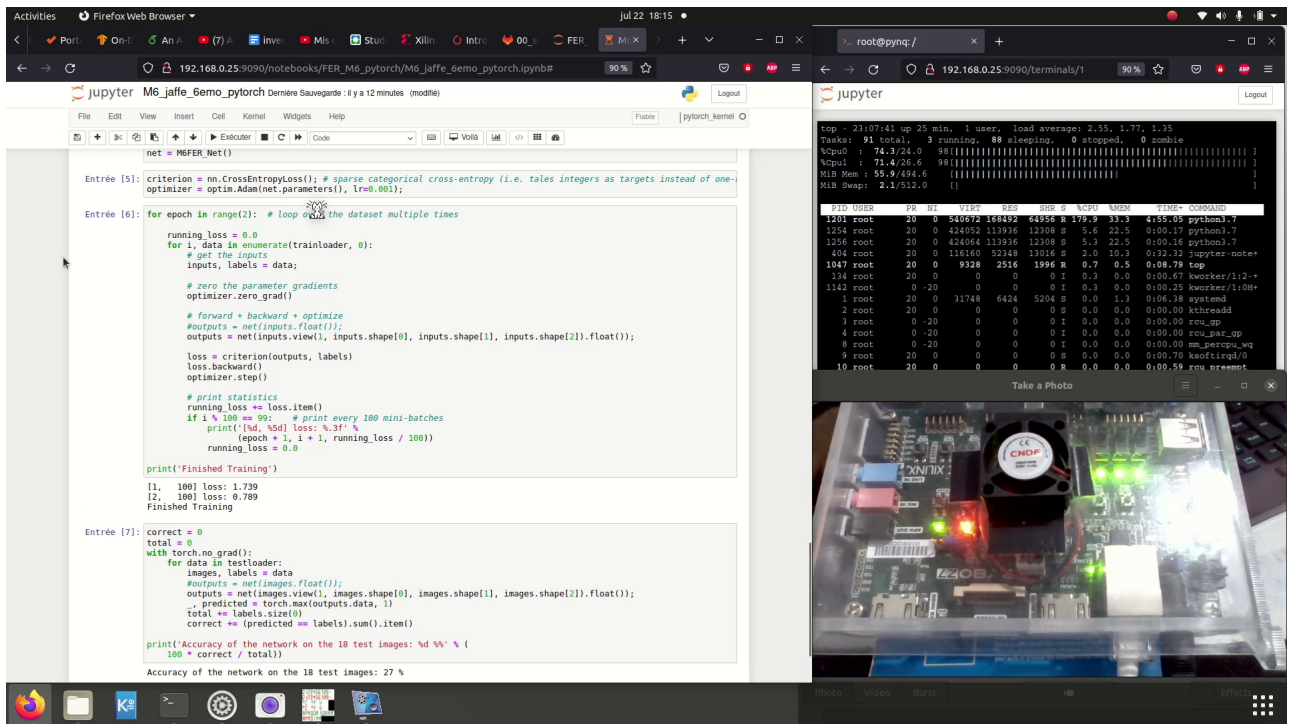


Figure C.5: M6 model trained with PyTorch 1.8 in Pynq 2.7.