Università degli Studi di Cagliari

MASTER DEGREE IN COMPUTER ENGINEERING, CYBERSECURITY AND ARTIFICIAL INTELLIGENCE

Report Final Lab - DNN-Based Image Classification on ARM

COURSE	STUDENT
Advanced Embedded Systems	Marco Ledda

A.A 2023/2024

Contents

Chapter 1

The System

1.1 Description of the Application

The objective of the final laboratory assignment is to deploy a micro-server on the Zybo Board Z7 (Fig. 1.1). This micro-server is built upon a Deep Neural Network designed to classify handwritten digits ranging from 0 to 9. The DNN used to implement the functionality is shown in the Fig. 1.2. The application accepts images of handwritten digits of dimension 28x28. They are images in greyscale, so we are dealing with data of dimension 1x1x28x28.

In essence, a typical Deep Neural Network comprises an input layer, one or more hidden layers, and an output layer. The specific DNN I employed includes:

- **Input Layer**: This layer, denoted as input data + flatten, is crucial because the neural network only accepts flattened arrays. One of its tasks is to transform an array into a shape of 1x28x28 (784 in length), catering to the network's requirements.
- **Two Hidden Layers**: Both are of the FC (Fully Connected) type and incorporate a ReLU activation function (Fig. 1.3). This function is employed

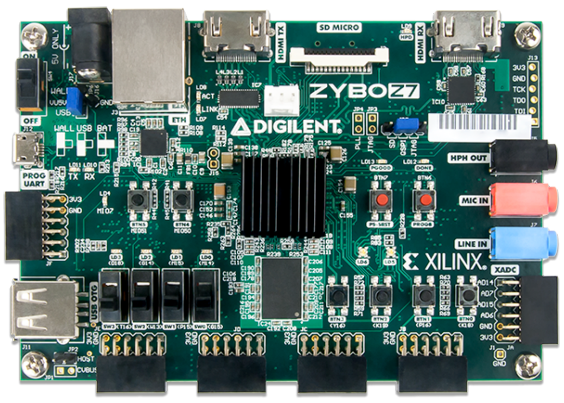


Figure 1.1: Zybo Z7 Board used for the development of the micro-server.

to classify a non-linearly separable input space. The first hidden layer takes a 784-element array as input and generates a 32-element output array, where 32 represents the number of hidden units in that layer. The second hidden layer takes a 32-element array as input, derived from the preceding layer, and produces a 16-element output array.

- **Output Layer**: The output layer is composed of a fully connected layer featuring a SoftMax activation function. This layer produces an array with ten elements, corresponding to its ten units. The purpose of this output is to present probabilities for each of the ten classes. The highest probability within the array indicates the predicted label for the input supplied to the machine.

Each unit of a layer performs, from a mathematical point of view, an output as a linear combination between inputs, weights and biases as:

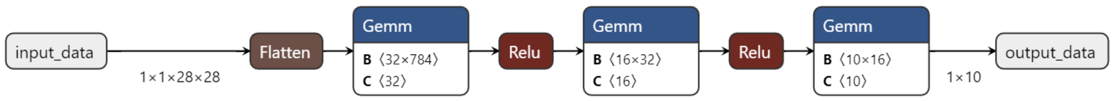


Figure 1.2: This is the DNN model assigned to me (Group 1). After the input layer, we have a first Forward Pass layer, then a ReLu layer, a second Forward Pass layer, then a ReLu layer and finally the output data, which is a vector made up of a score of accuracy for each label (0-9) present in the dataset.

$$f(x) = \sum_{i=1}^{n} w_i x_i + b_i \tag{1.1}$$

where w_i is a component of the vector of weights, b_i is a component of the vector of biases and x_i is a component of the current activation vector.

In the given application, the ReLu forward is used as an activation function. The ReLU (Rectified Linear Unit) activation function is characterized by

$$f(x) = \max(0, x) \tag{1.2}$$

as illustrated in Fig. 1.3. Its purpose is to eliminate consideration of negative inputs while linearizing the positive ones. Compared to alternative activation functions, its advantage lies in its simplicity, contributing to more effective network training.

Before the Deep Neural Network can effectively classify images, it must be trained, typically utilizing a backward propagation algorithm. This training process enables the network to learn the appropriate weights, biases, and consequently, discern patterns within the provided training input. The assignment provides the necessary weights and biases, rendering the network prepared for classification tasks. In summary, the objective is to take weights, biases, and images as input, apply them to the algorithmic implementation of the DNN, and compute fundamental performance metrics such as execution time and accuracy of the network.

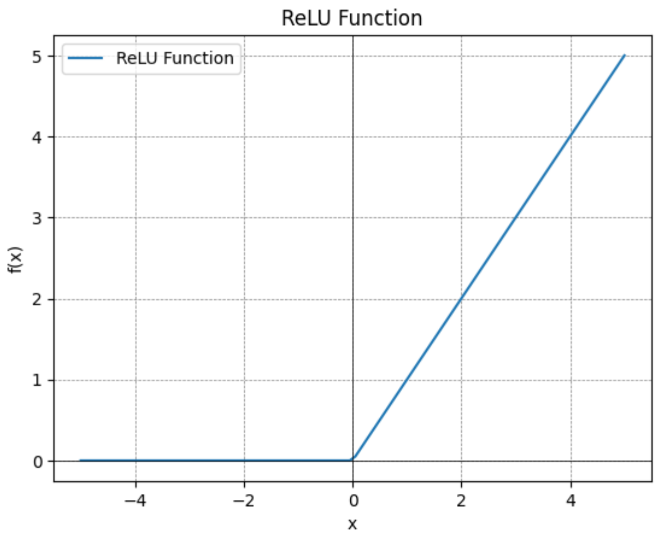


Figure 1.3: Graphical representation of the Rectified Linear Unit in the plane.

1.2 System characteristic

This section introduces the system's characteristics, including the main on-chip and off-chip components used to implement the micro-server application. The Zynq system design is represented in the Fig. 1.4.

On-chip components. If we look at the Fig. 1.4, we can see how the Zynq system is made up. The most important components are the Dual-core ARM Cortex A9 processor, the two-level cache and the I/O peripherals in order to allow the communication between the processor and the external enviroment. The UART (Universal Asynchronous Receiver-Transmitter) is a fundamental block which allows communication with the Software Development Kit and data transmission up to 1 Mb/s.

Off-chip components. Regarding the off-chip components we have:

- **RealTerm**: a really useful software used to employ the serial communica-

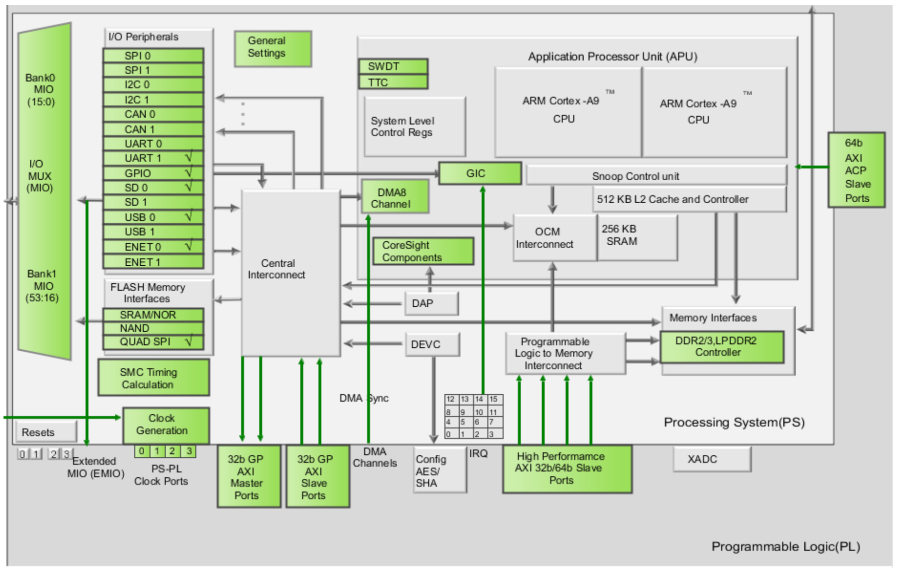


Figure 1.4: Zynq system design. The picture can be seen in the block design of Vivado.

tion. With this tool I've been able to send weights and bias files for each layer of the network and the images to fed the network and get the classification label.

- **USB Port**: used by RealTerm to connect the board with the local machine.
- **HexEditor**: hexeditor is a useful tool which allowed me to visualize the content of the weights, bias and images binary file. The file provided are in 16-bits representation.

1.3 The Application

The application is developed for recognizing handwritten digits (an example in Fig. 1.5) taken from the common MNIST dataset. When we start the application, it asks for weights and biases files, provided as binary files. Then, the application requires the digit to classify. Weights and biases are needed by the Deep Neural

Figure 1.5: Example of handwritten digit.

Network in order to fulfill the classification. The micro-server is implemented using Xilinx SDK, a Software Development Kit for embedded software applications whose targets are Xilinx Embedded processors. The SDK allows to write C code implementing the application and also to understand the assembly code produced by the compiler, which can be suitably configured based on the application to write.

The code with which I implemented the micro-server defines arrays for weights and biases of type DATA. The readDATA() function reads two bytes at a time and composes the data by shifting the most significant byte by 8 positions. I have also defined a wrapper called createDataArray(), which calls the readDATA(), to create the array with the bytes read from the UART implementing a polling procedure. This function was used only to create data structures for weights and biases. To create the data structure for the image, I defined a wrapper called createImage(). In the main function, once the appropriate data structures for weights and biases are created, a while loop is initialized. This way, the application continuously accepts a new image for classification. The application is implemented by following the mathematical concepts presented in 1.1, through functions FC_forward, relu_forward, and resultProcessing whose aim is to compute the probability for each of the ten classes and return the predicted label.

Chapter 2

Baseline Performance

2.1 Performance measurement setup

In this section we formalize the measurement setup, focusing on the following parameters:

- **GOPS/s**: which are the GigaOperations per second, performed by the board;
- **Bitrate**: related to send and receive operations;
- **Execution time**: expressed in microseconds in the results tables, for both the entire micro-server and individual layers of the Deep Neural Network.
- **Accuracy of the network**: refers to how well the model is able to correctly predict or classify the input data.

GigaOperations per seconds. The GOPS/s represents the number of effective GigaOperations per second made by the board. It is computed as:

$$\text{GOPS/s} = \frac{2 * \text{Input size} * \text{Output size}}{\text{time}} * 10^{-9} \tag{2.1}$$

Input size and Output size parameters refer to the sizes of the vectors taken into account when the computation between vectors is comptuted. The multiplication by 2 is done in order to consider the operations on the mac variable in the attached code.

Bitrate. The bitrate for each image received is computed as:

$$\text{bitrate}_{receive} = \frac{\text{Bit of the image}}{\text{time}} \quad [b/s] \tag{2.2}$$

On the other hand, the bitrate for each image sent is computed as:

$$\text{bitrate}_{send} = \frac{\text{Printed Bits}}{\text{time}} \quad [b/s] \tag{2.3}$$

Execution time. The execution time for each layer is computed as:

$$\text{time} = \frac{\text{Clock cycles}}{667 * 10^6} \quad [s] \tag{2.4}$$

where 667 MHz is the clock frequency of the CPU. We have to take into account that the clock cycles can be easily computed as:

$$\text{Clock cyles} = t_{end} - t_{start} \tag{2.5}$$

These measurements of time are computed using the built-in instructions provided by the library xtime_l.h. The final overall execution time will be the sum of all the execution time of each layer of the Deep Neural Network.

Accuracy of the network. Finally, we have to consider also a metric to compute the accuracy of the system in the classification task. Accuracy is typically measured as the ratio of correctly predicted instances to the total number of instances in the dataset. It is one of the common evaluation metrics for classification tasks. Here the formula used:

$$\text{accuracy} = \frac{\text{Samples correctly classified}}{\text{Total number of test samples}} * 100 \quad [\%] \tag{2.6}$$

2.2 Baseline configuration

The baseline configuration of my compiler is:

```
-c -fmessage-length=0 -MT"$@" -mcpu=cortex-a9 -mfpu=vfpv3
-mfloat-abi=hard
```

- **-mcpu=cortex-a9**: it sets the available instruction set based on the available CPU.
- **-mfpu**: it specifies the floating-point unit available on the hardware. It is used when using compiler optimization efforts.
- **-mfloat-abi=hard**: it specifies which floating-point ABI to use. "Hard" allows the generation of floating-point instructions

2.3 Execution times

As shown in Table 2.1, the application is run with no compiler-level optimizations (-O0) and using optimization (-O2). The software used to send weights, biases and images is RealTerm. The baudrate set is @115200. As expected, each layer of the DNN has been executed in a smaller time with respect to the previous one, as the amount of data to compute is reduced with each step. On average, the bitrates for the receiving operation is around 15510 b/s, while the sending operation operates at approximately 59015 b/s. The effective bitrate is intricately linked to the specific image being processed and the type of string intended for printing. GOPS/s are reported in Table 2.2 Regarding the accuracy of classification, 90% of accuracy has been achieved during tests.

Layer	**-O0 Setting [μs]**	**-O2 Setting [μs]**
Receive	808734.12	801046.75
FC0	1358.34	197.37
ReLu0	1.48	0.33
FC1	29.3	4.88
ReLu1	0.82	0.17
FC2	10.01	1.73
Result Processing	1.63	1.23
Overall response time of the DNN	1401.58	205.71
Send	2168.92	2168.92
Overall response time including receive and send	812304.62	806421.38

Table 2.1: Execution times - No Optimization, -O0 and -O2 settings

	-O0 Setting	**-O2 Setting**
GOPS/s	0.0369	0.254

Table 2.2: Gigaoperations per second in -O0 setting and -O2 setting for baseline implementation.

2.4 Memory footprint

The linker script is a script written in a specific syntax that defines the layout and behavior of the final executable or library generated by a linker during the compilation process. Basically, it controls the location of each piece of the executable in the memory. During the development of the project, there was the need to increment the size of the stack. The Zybo Z7 board started from a size of 0x2000 and I incremented it to 0xB000 (some trials have been done in order to reach this value). I didn't change the heap size since I didn't use dynamic allocation in my implementation. Only static allocation was used. The main reason for increasing the stack is that there were already many statically declared arrays in the project base.

If we look at the linker script in the folder **../src/lscript.d** of the project folder, we can see that all the section (from .text to .stack) are mapped into the ps7_ddr_0 register in memory (in the specific, it is a DDR SDRAM - Double Data Rate Synchrounous RAM).

Chapter 3

Optimizations

NEON technology represents ARM's integration of the ARM's Advanced Single Instruction, Multiple Data (SIMD) architecture. This design enables the creation of instructions capable of processing multiple data points concurrently, optimizing execution time through parallelization compared to non-SIMD instructions. There are many ways to make use of NEON technology. Here below, the two optimizations implemented :

- Auto-vectorization;
- Neon Intrinsics;

3.1 Autovectorization for NEON

I've implemented a performance enhancement in the code by tapping into NEON technology. The key improvement stems from the utilization of auto-vectorization, a technique employed by the compiler. This process involves scrutinizing loops within the code and intelligently converting them from a less efficient scalar implementation (one operation at a time) to a more optimized vectorized version (more operations at a time).

With the incorporation of the NEON engine, the compiler can seamlessly parallelize operations during the generation of assembly code through GCC. This means that NEON has the ability to significantly expedite the execution of operations directly, resulting in an overall acceleration of the code's performance. In simpler terms, the code should now runs faster and more efficiently, thanks to the synergistic application of auto-vectorization and the capabilities of the NEON engine. The following flags have been set:

```
-c -fmessage-length=0 -MT"$@" -mcpu=cortex-a9 -mfpu=neon
-mfloat-abi=hard -std=c99 -mvectorize-with-neon-quad
-ftree-vectorize
```

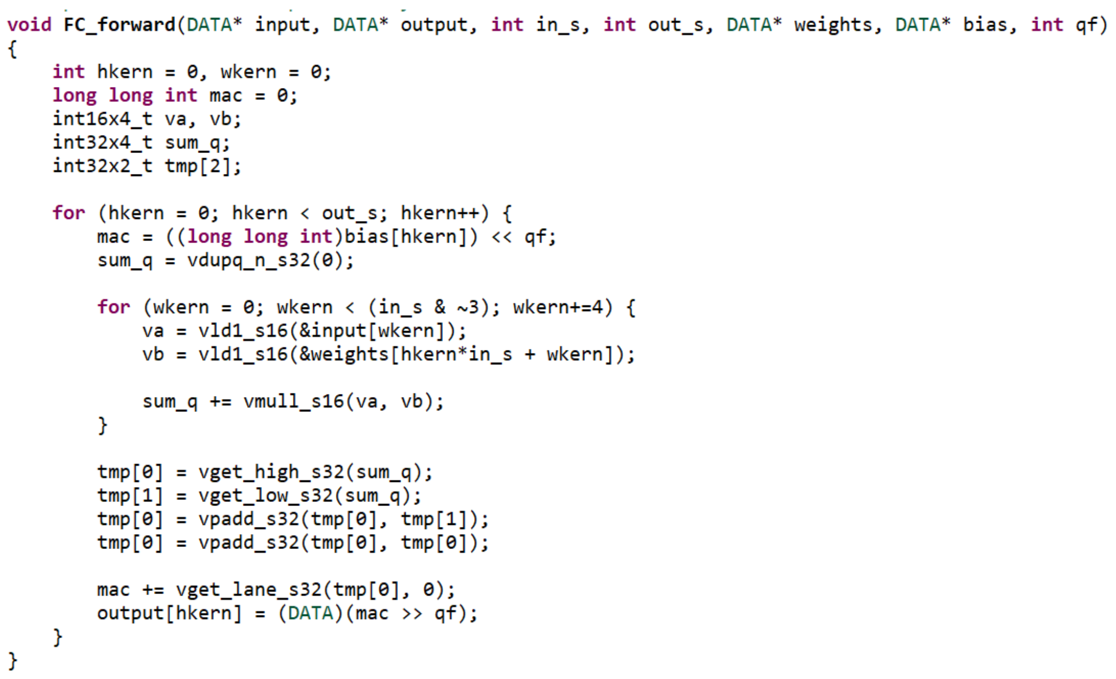
The main difference between this configuration and the basic compiler is the floating-point unit which is now NEON. Morevorer, three flags have been specified:

- -std=c99: introduces new features used by the NEON engine;
- -ftree-vectorize: enables vectorization of C code;
- -mvectorize-with-neon-quad: the vectorization is done with quad words instead of double words;

Actually, the most efficient method to make use of NEON's capabilities is to manually craft NEON assembler code, but this approach is time-consuming and challenging. A more balanced alternative involves utilizing NEON's vectorization through intrinsics.

3.2 Autovectorization for NEON with Intrinsics

The second approach to optimize the execution time of the DNN is to use the same underlying idea, modifying the FC_forward() calling the NEON functions provided by the NEON documentation. The access to these functions is really simple. Here the attempt of implementation:



```
void FC_forward(DATA* input, DATA* output, int in_s, int out_s, DATA* weights, DATA* bias, int qf)
{
    int hkern = 0, wkern = 0;
    long long int mac = 0;
    int16x4_t va, vb;
    int32x4_t sum_q;
    int32x2_t tmp[2];

    for (hkern = 0; hkern < out_s; hkern++) {
        mac = ((long long int)bias[hkern]) << qf;
        sum_q = vdupq_n_s32(0);

        for (wkern = 0; wkern < (in_s & ~3); wkern+=4) {
            va = vld1_s16(&input[wkern]);
            vb = vld1_s16(&weights[hkern*in_s + wkern]);

            sum_q += vmull_s16(va, vb);
        }

        tmp[0] = vget_high_s32(sum_q);
        tmp[1] = vget_low_s32(sum_q);
        tmp[0] = vpadd_s32(tmp[0], tmp[1]);
        tmp[0] = vpadd_s32(tmp[0], tmp[0]);

        mac += vget_lane_s32(tmp[0], 0);
        output[hkern] = (DATA)(mac >> qf);
    }
}
```

Figure 3.1: FC_forward() intrinsics approach

NEON typically works on 128-bit registers. This implementation uses two data types: int16x4 and int32x4. The first type contains 4 values of 16 bits, while the second one contains 4 values of 32 bits. This attempt of implementation works on 4 values at a time, thus the vector is separated into 4 element groups and iterated through a for loop. Inside the inner for loop, we can see the function vld1_s16() which loads 4 16-bit signed values in one int16x4 register. Then,a multiplication among vectors is performed through the function vmull_s16(va, vb). This function makes a multiplication of two signed integer values of the vectors taken as input. Then, it places the results in a vector of int32x4 type. At the end of the inner loop, we exploit a temporary variable of type int32x2 to store the high and the low part of the data accumulated in the sum variable. The function vpadd_s32() performs the pairwise add among the high and the low part of the data accumualtor. Finally, the function vget_lane_s32() extracts the final result from the temporary storage. It duplicates the element specified at the provided index. It does not care which index is provided because both cells contain the same result.

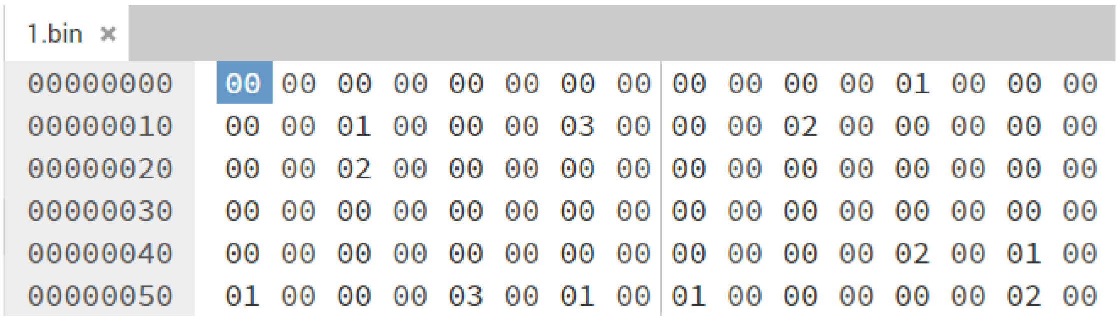
3.3 Reduced data

This optimization centers around the concept of minimizing the number of bits employed to represent data. The utilization of an hexadecimal editor proved instrumental in this context. Upon inspecting an image file, a consistent pattern emerges – the Most Significant Byte (even column, little-endian format) consistently holds a value of 00 (Fig. 3.2). This observation means that the pixels of the image are invariably positive. Exploiting this characteristic, it becomes feasible to employ a storage strategy that utilizes only 8 bits to represent the pixel values without sacrificing sign information. Essentially, since the Most Significant Byte is consistently 00, the assumption can be made that the pixel values are always positive. This reduction in the number of bits used for representation not only conserves storage space but also streamlines processing without compromising the integrity of the image data. For instance, if we try to open the image for digit 2 (2.bin), we can notice that we have a pixel as:

0x0010 = 0b 0000 0000 0001 0000

in which the sign bit (0) is extended to fill 16 bits.

A comparable strategy can be applied to the weights and biases files. When examining these files, a distinctive pattern emerges - the Most Significant Byte (even column, little-endian) consistently holds values of either 00 or FF (Fig. 3.3). This pattern implies that the weights or biases could be either positive or neg-



```
1.bin ×
00000000  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00
00000010  00 00 01 00 00 00 03 00 00 00 02 00 00 00 00 00
00000020  00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040  00 00 00 00 00 00 00 00 00 00 00 00 02 00 01 00
00000050  01 00 00 00 03 00 01 00 01 00 00 00 00 00 02 00
```

Figure 3.2: Example of digit.bin file opened with Hex Editor. As we can see, the MSB (even column, little-endian format is employed) is always 00.

ative. However, it's crucial to note that in this scenario, the sign information cannot be disregarded. To address this, the Q1.7 representation is employed. The Q1.7 representation is characterized by a structure where 1 bit is dedicated to sign information, and the remaining 7 bits represent the value. This approach allows for the representation of both positive and negative values. For instance, if we try to open the file of the first layer's weight, we can notice that we have values like:

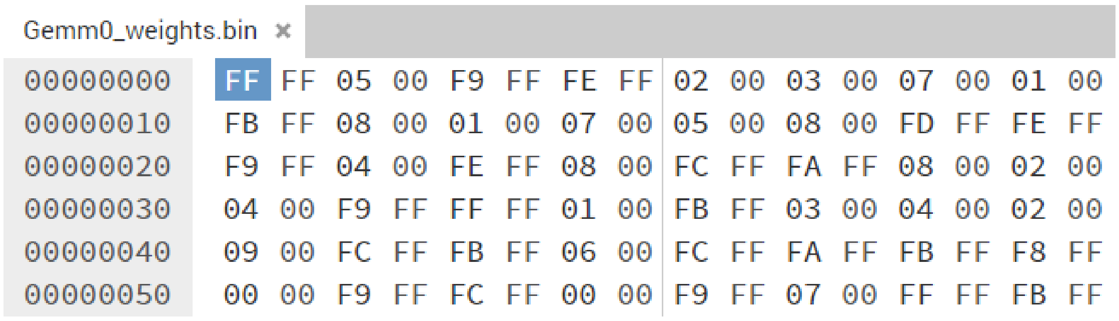
0xFFF9 = 0b 1111 1111 1111 1001

in which the first 8 bits starting from left is the sign (negative in this case) and the last 8 bits represent the value. The final objective is to have this notation:

0xFFF9 = 0b 1 1111100

where the first bit from left represent the sign and the remaining 7 bits represent the value.

Nevertheless, it's essential to acknowledge that there is a potential risk of information loss concerning weights and biases. The utilization of Q1.7 representation might compromise the accuracy of the network due to the limited bit precision,


Gemm0_weights.bin

```
00000000  FF FF 05 00 F9 FF FE FF 02 00 03 00 07 00 01 00
00000010  FB FF 08 00 01 00 07 00 05 00 08 00 FD FF FE FF
00000020  F9 FF 04 00 FE FF 08 00 FC FF FA FF 08 00 02 00
00000030  04 00 F9 FF FF FF 01 00 FB FF 03 00 04 00 02 00
00000040  09 00 FC FF FB FF 06 00 FC FF FA FF FB FF F8 FF
00000050  00 00 F9 FF FC FF 00 00 F9 FF 07 00 FF FF FB FF
```

Figure 3.3: Example of weight.bin file opened with Hex Editor. As we can see, the MSB (even column, little-endian format is employed) is always either 00 or FF.

and as a result, careful consideration should be given to balancing the need for compression with maintaining sufficient accuracy in the neural network.

Chapter 4

Performances of the Optimizations

The results of NEON optimizations are presented in Table 4.1, detailing both auto-vectorization and intrinsic approaches. The baud rate is configured at 115200. Although auto-vectorization yielded satisfactory results, the manual optimization using NEON intrinsics led to an inferior overall performance, specifically in the overall response time of the DNN. This discrepancy is likely attributed to a sub-optimal implementation.

Regarding optimization with reduced data, several have been done attempts:

First Attempt. The first attempt was made through the implementation of a function FC_forward() that takes in input data of 16-bits converted into 8 bits through the normalization function. Also weights and biases are passed as 8 bits making a conversion from 16 bits to 8 bits. This introduced an improvement in terms of memory footprint but not in terms of execution times. In fact, the receive time is always the same and not halved. However, the classification accuracy dropped significantly making the DNN inconsistent, as they only managed to correctly classify 2 out of 10 samples.

Second Attempt. To avoid the problem of the accuracy, a second attempt has been done. The second attempt was to implement two distinct functions called FC_forward_0() and FC_forward(). The first is used only in the first layer of the DNN, the second in the subsequent layers. FC_forward_0() holds weights, biases and activations in 8 bits while FC_forward() reports activations in 16 bits. This is most likely due to anomalous behavior in the accumulation variable, because after the first Fully-Connected Forward 8 bits are no longer sufficient for the output values. In this case, the accuracy drops to 70%, so the classification results remain at least reliable.

The results of reduced data optimization are presented in Table 4.2, detailing both the first and the second attempt explained above.

Layer	**Auto-Vectorization [μs]**	**NEON Intrinsics [μs]**
Receive	810439.5	804512.89
FC0	92.16	105.71
ReLu0	0.11	0.13
FC1	3.16	3.26
ReLu1	0.01	0.01
FC2	1.43	1.40
Result Processing	1.16	1.21
Overall response time of the DNN	98.03	111.72
Send	2168.16	2168.87
Overall response time including receive and send	810537.53	806793.48
GOPS/s	0.544	0.474

Table 4.1: Execution times - Auto-vectorization and NEON Intrinsics comparison

Layer	**First attempt [μs]**	**Second attempt [μs]**
Receive	808221.87	408221.87
FC0	158.21	70.13
ReLu0	0.28	0.07
FC1	4.42	2.46
ReLu1	0.18	0.01
FC2	1.71	0.73
Result Processing	1.04	1.31
Overall response time of the DNN	165.84	74.71
Send	2168.93	2168.97
Overall response time including receive and send	810556.64	810700.12
GOPS/s	0.317	0.715

Table 4.2: Execution times - Reduced data 8 bits, First and Second attempt

Chapter 5

Conclusions

In this final report, I have detailed the deployment of a Deep Neural Network (DNN) for the classification of hand-written digits on the Zybo board. The DNN functions as a micro-service, accepting weights, biases, and images in the form of binary files as inputs.

The findings reveal the feasibility of crafting a neural network specifically tailored for execution in an environment with restricted resources, be it computational capabilities or memory constraints. Furthermore, this report illustrates the potential for optimization by adjusting either the properties of the GCC arm compiler (via optimization efforts and ARM NEON auto-vectorization) or by managing the memory footprint. The findings show that it is possible to create a Neural Network that is designed to work in an environment with limited resources, such as limited computing power or memory. This report also demonstrates that optimization can be achieved by managing the memory usage. The settings of the GCC Arm compiler helped me to improve the optimization effort through NEON auto-vectorization.

To find the best way, I compared different measurements. The best strategy depends on specific needs. For example, if we need to handle a lot of information, we might need to classify it quickly. On the other hand, if we don't have a lot of memory, we might need to use less detailed information but still classify accurately.