Advanced Embedded Systems

Answers for Assignment 3

SILVIA LUCIA SANNA – 70/90/00053

Step 1,2 – Set the environment and modify network topology

The first thing I have to do in this assignment is to set properly the environment where to execute the code. So, I execute the Azure Virtual Machine and open "QTerminal" in which I have to call the SDK using command "xsdk".

Once the SDK opened, I created a new project in the Desktop called "assignment 3" having a Linux platform and Cortex A53 processor. As I create the project, I have a "helloworld.c" file that I must delete to not have a redundant main as I have to write the codes implementing the DNN. I will replace the helloworld file with those I need to execute the DNN, using the ones you gave us but changing in the right way the network implementation in mnist_only_FC.c file with the configuration I can see in my group file "onnx_model.onnx" using Netron browser.

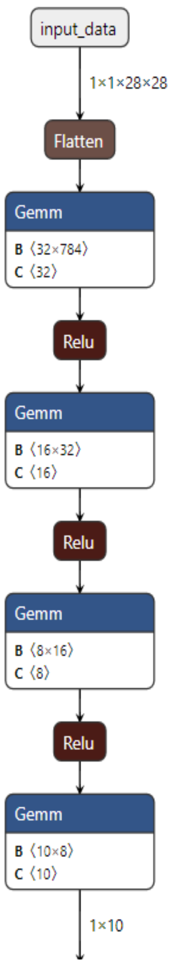


Figure 1: old network architecture (version 1)

So, in the "mnist_only_FC.c" file I have to change the input and output numbers of each layer and add a Relu3 layer and a Gemm3 layer, changing properly the connection between these layers. Here the results will be taken from Gemm3's output instead of from Gemm2 as in the example you gave us.

In this network there is a problem, Gemm3 receives as input a data of size 8 but produces as output size 10 which is incorrect: you cannot have a little input and big output dimension. In fact, as I will explain later, this will lead to misclassification.

Changing network settings with the correct ones, the results improved. To see the new network topology, I downloaded the fixed version from GitHub

(https://gitlab.com/paololoceri/aes_mnist_assigment/-/tree/master/group_3_fix)

and I changed the file mnist_only_FC.c with the right input size for each layer, total weights for each layer and bias for every layer. Then, I also added the new weights and bias binary files in the Debug directory where the elf file was.

The network I used in the next tests was the following one:

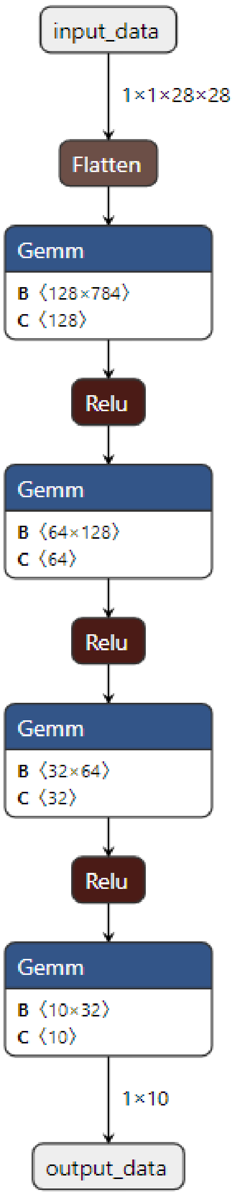
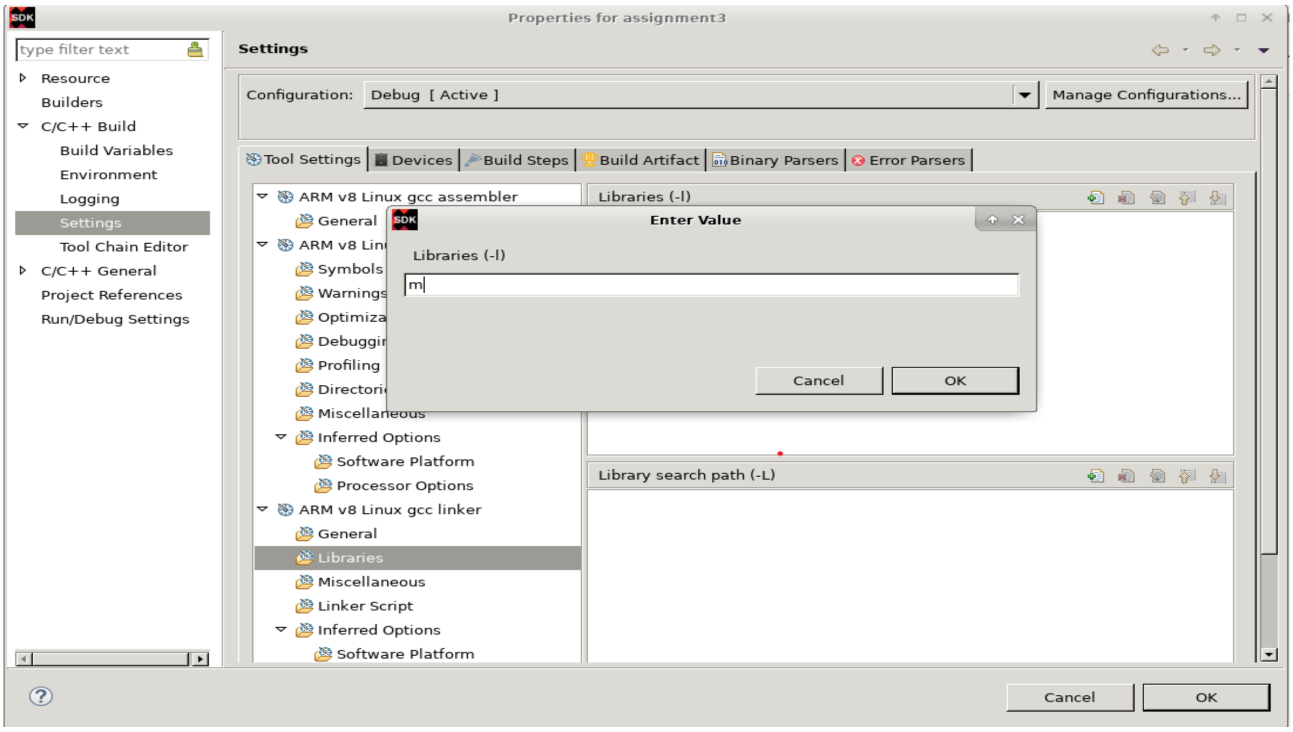


Figure 2: new network topology

In the codes I also have to add the math library, using the following procedure: right click in the project folder → C/C++ Settings which will prompt the following window. In the settings I have to select "ARM v8 Linux gcc linker" and its "Libraries". In the above window (libraries -l) I have to add the command "m" which stands for the math library. Then click on OK and the setting is done.

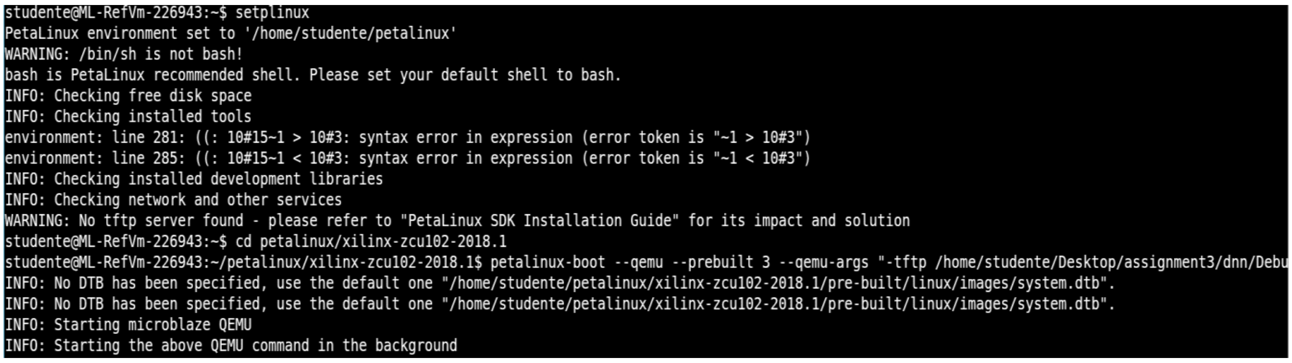


Configuration: Debug [Active]

Libraries (-l): m

Figure 3: how to set math library

In the project I will not use the processor A53 but on a virtual instance based on QEMU. To do this I have start the virtual processor on QEMU using the following commands:

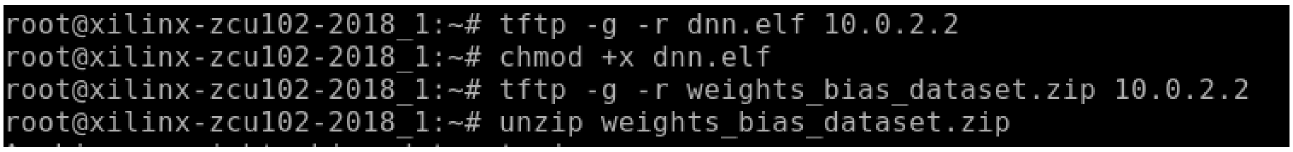


```
studente@ML-RefVm-226943:~$ setplinux
PetaLinux environment set to '/home/studente/petalinux'
WARNING: /bin/sh is not bash!
bash is PetaLinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
environment: line 281: ((: 10#15~1 > 10#3: syntax error in expression (error token is "~1 > 10#3")
environment: line 285: ((: 10#15~1 < 10#3: syntax error in expression (error token is "~1 < 10#3")
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "PetaLinux SDK Installation Guide" for its impact and solution
studente@ML-RefVm-226943:~$ cd petalinux/xilinx-zcu102-2018.1
studente@ML-RefVm-226943:~/petalinux/xilinx-zcu102-2018.1$ petalinux-boot --qemu --prebuilt 3 --qemu-args "-tftp /home/studente/Desktop/assignment3/dnn/Debu
INFO: No DTB has been specified, use the default one "/home/studente/petalinux/xilinx-zcu102-2018.1/pre-built/linux/images/system.dtb".
INFO: No DTB has been specified, use the default one "/home/studente/petalinux/xilinx-zcu102-2018.1/pre-built/linux/images/system.dtb".
INFO: Starting microblaze QEMU
INFO: Starting the above QEMU command in the background
```

Figure 4: set the environment

Where "setplinux" is to set environment variables, cd petalinux is to move into petalinux project folder where I execute petalinux-boot to launch emulator shared folder. Than with tftp I make a connection between the Azure machine and the FPGA virtual environment.

After this I have to load the elf file and the folders with datasets and binaries in the virtual processor, using the following commands, giving also permissions to elf file to be executed.



```
root@xilinx-zcu102-2018_1:~# tftp -g -r dnn.elf 10.0.2.2
root@xilinx-zcu102-2018_1:~# chmod +x dnn.elf
root@xilinx-zcu102-2018_1:~# tftp -g -r weights_bias_dataset.zip 10.0.2.2
root@xilinx-zcu102-2018_1:~# unzip weights_bias_dataset.zip
```

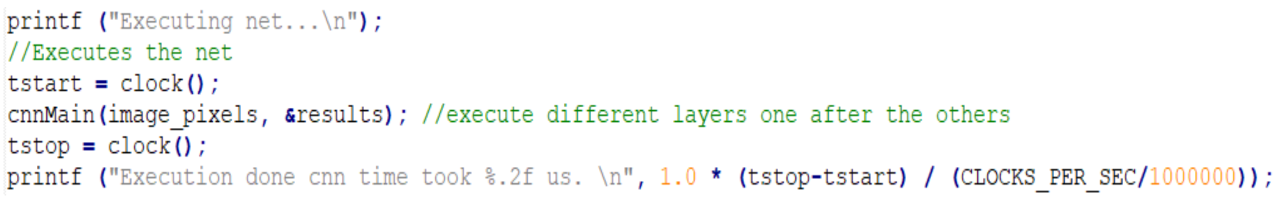
Figure 5: make the connection with tftp

Whenever I shut down the Azure Virtual Machine and I want to execute the code, I have to recreate the virtual instance of the processor with the setplinux commands above. The command to change elf permissions' rights is done only the first time in the session, while the tftp connection to the elf file is done every time I change something in the code, saving and building it.

The weights_bias_dataset.zip is a compression of the directories containing the dataset in .txt format and the weights and biases binaries for each Gemm layer. This compressed directory is in the same directory of the elf file, so the Debug directory of the project, to which I made the tftp connection between the Azure VM and the emulation of the FPGA.

Step 3 – Measure execution time

To measure execution time I added library "time.h", defined 2 variables of type clock (the first one for start time and the second one for stop time) and took the time before and after the cnnMain function (the one that creates the network structure by calling appropriate functions for each layer filling).



```
printf ("Executing net...\n");
//Executes the net
tstart = clock();
cnnMain(image_pixels, &results); //execute different layers one after the others
tstop = clock();
printf ("Execution done cnn time took %.2f us. \n", 1.0 * (tstop-tstart) / (CLOCKS_PER_SEC/1000000));
```

Figure 6: measure execution time of cnnMain (network execution) in main.c

To know how much time did the cnnMain function spent to be executed, I made the difference between stop and start time and convert this result in microseconds.

I measured execution time for all 9 digits and it was between 70 ms and 181 ms.

Step 4 – Classify digits 0, 3, 5

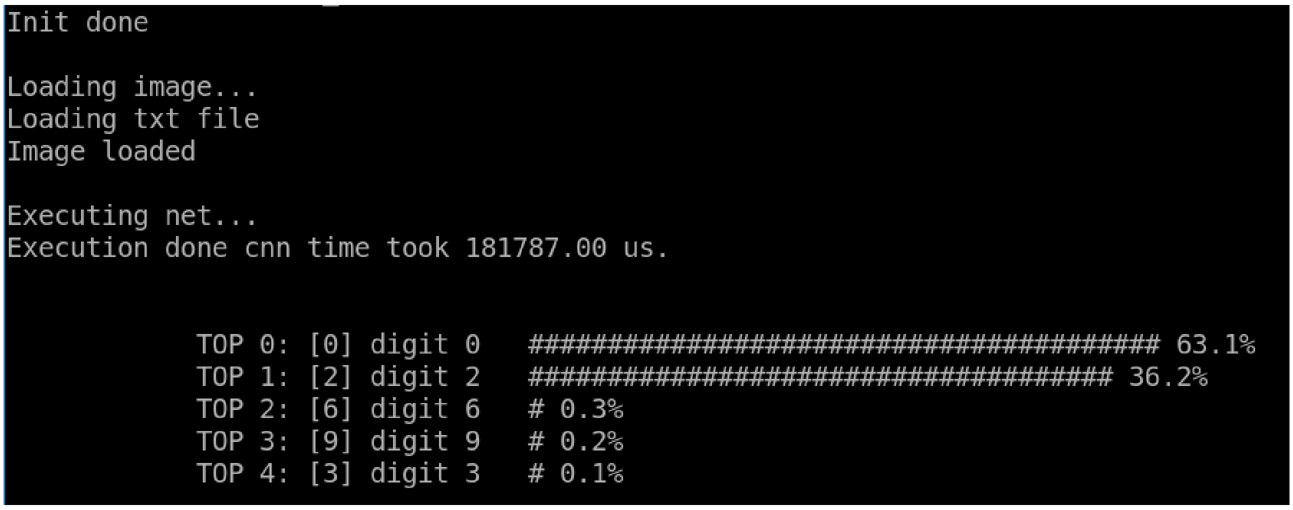
The aim of this assignment was to do some classification using Deep Neural Networks in a FPGA and to talk about this task I will shortly explain how this must be done.

First of all in a classification problem you need a dataset, a set of different samples of the same type of the object you want to classify. In this assignment I had to classify handwritten digits and MNIST dataset was used.

After that is important to have a right structure of the network, which is a set of different elements (perceptrons) that linked in the proper way are able to distinguish between samples coming from different classes (samples of different types). The structure of the network was given to us and it is the one I explained in the Step 1 task. In the network it is essential to establish the right weights and bias which can be seen as the criteria used by the network to find the corresponding class of the sample received in input.

The network I used was pre-trained, what does it mean? I used weights that were optimal in a previous network classification, instead of searching them randomly and changing them in different iterations until I reached the less mistakes. It is called pre-trained because in the first step the network I use is pre-trained, so it was just trained in another task. This approach is better because instead of doing some random research and improve that value, the likelihood that some applications have similar weights if the two datasets and experiments have enough features/characteristics in common is high enough. Instead, if you already know the best weights, you use them.

After that, now everything was ready to start classifying the last three digits of my student ID which are 0,3,5.



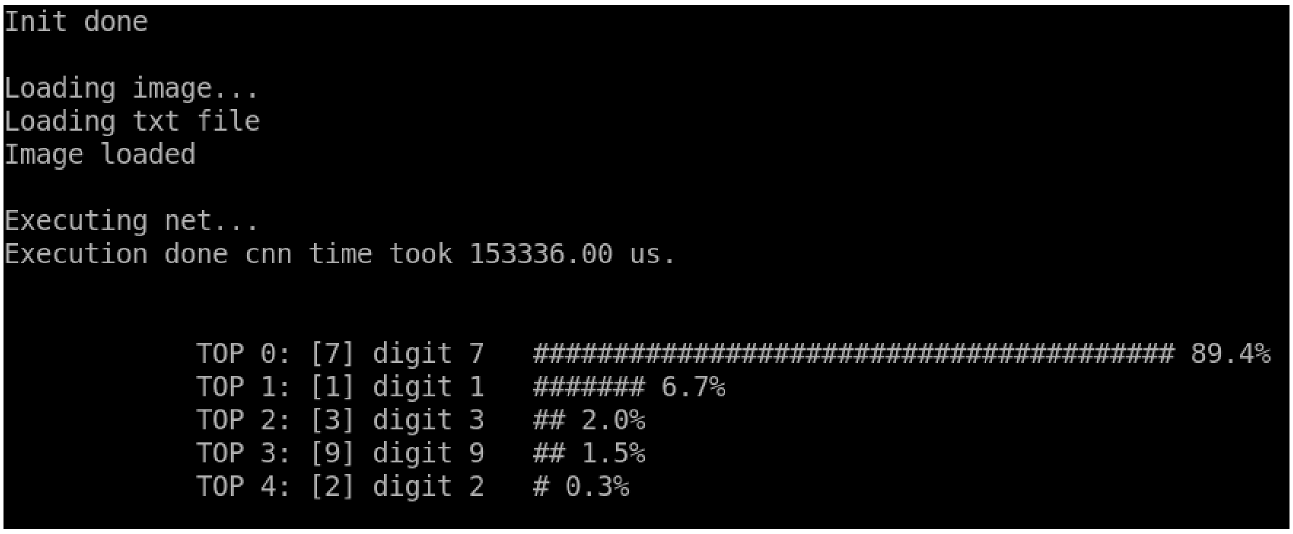
```
Init done

Loading image...
Loading txt file
Image loaded

Executing net...
Execution done cnn time took 181787.00 us.


            TOP 0: [0] digit 0   ######################################## 63.1%
            TOP 1: [2] digit 2   ##################################### 36.2%
            TOP 2: [6] digit 6   # 0.3%
            TOP 3: [9] digit 9   # 0.2%
            TOP 4: [3] digit 3   # 0.1%
```

Figure 7: performances for digit 0



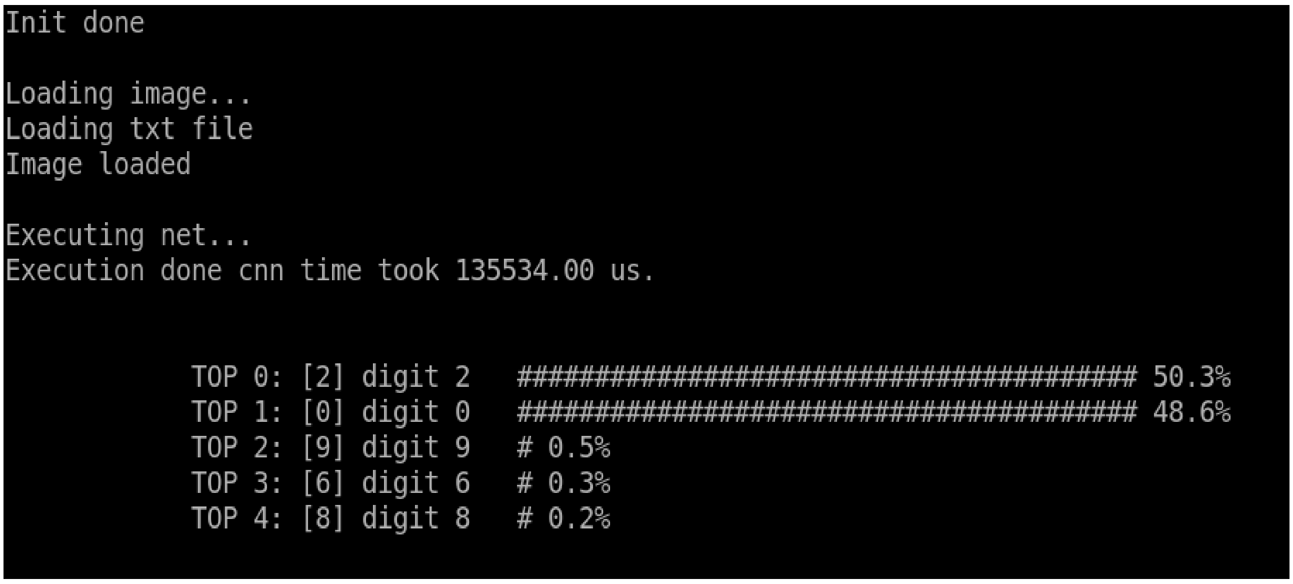
```
Init done

Loading image...
Loading txt file
Image loaded

Executing net...
Execution done cnn time took 153336.00 us.


            TOP 0: [7] digit 7   ######################################## 89.4%
            TOP 1: [1] digit 1   ####### 6.7%
            TOP 2: [3] digit 3   ## 2.0%
            TOP 3: [9] digit 9   ## 1.5%
            TOP 4: [2] digit 2   # 0.3%
```

Figure 8: classification output for digit 5



```
Init done

Loading image...
Loading txt file
Image loaded

Executing net...
Execution done cnn time took 135534.00 us.


            TOP 0: [2] digit 2   ######################################## 50.3%
            TOP 1: [0] digit 0   ######################################## 48.6%
            TOP 2: [9] digit 9   # 0.5%
            TOP 3: [6] digit 6   # 0.3%
            TOP 4: [8] digit 8   # 0.2%
```

Figure 9: classification performance for digit 3

From the results you can easily see that number 0 is the most correct value to be classified with an accuracy of 63.1%. Numbers 5 and 3 instead are the most wrong classified ones, in fact in both of them the right number is not even in the list of the most similar ones. These bad results can be done by an insufficient training in the network (weights are not the best ones for this numbers, so we must retrain the network looking for the optimized values) or even the topology of the network is incorrect, we should need one or more layers.

Step 6 – Optimize ReLU and Linear Functions

I decided to implement also the bonus task, so optimize the ReLU and Linear functions using the parallelization to speed up the computation of:

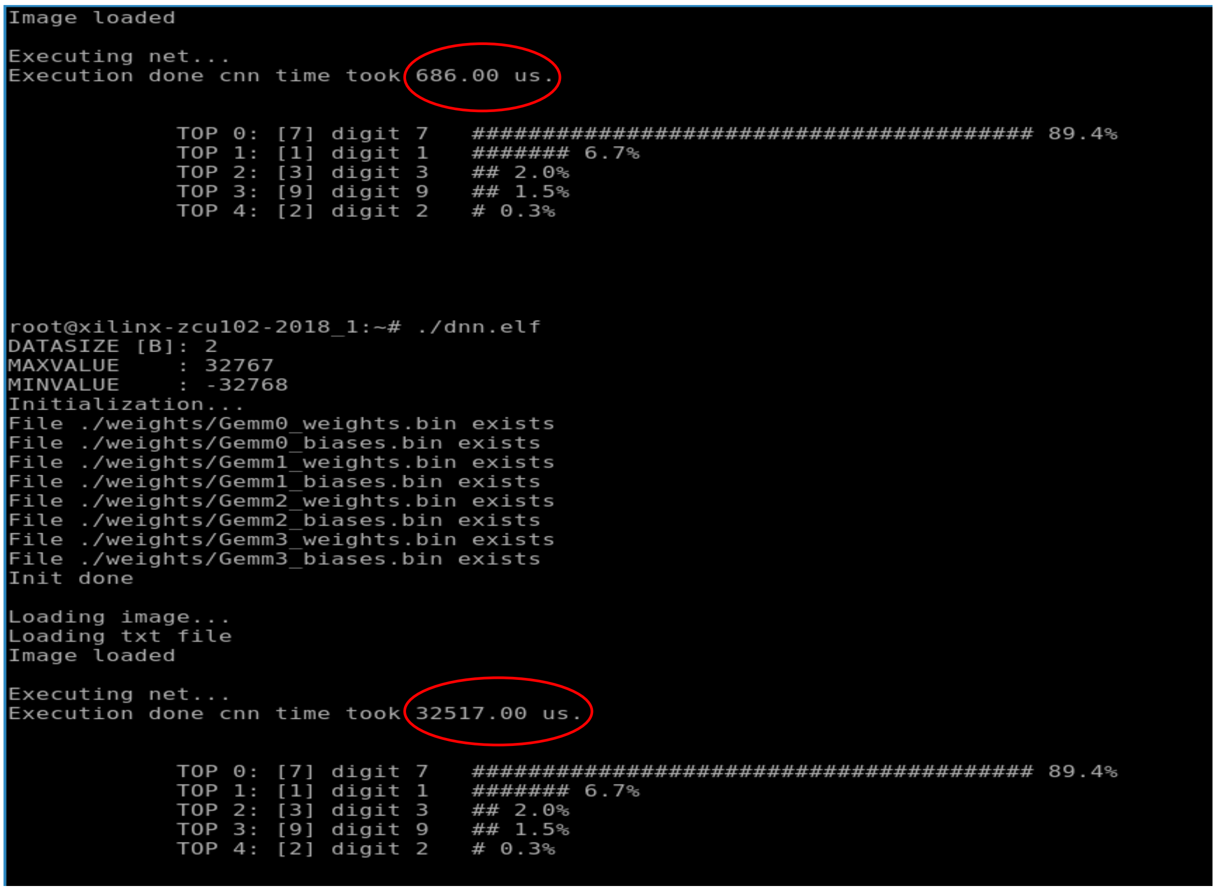
- ReLU function: I have to improve the execution in the assignment of a precise value with respect to its greatness to a threshold value, so the fact to rectify or not a value.
- Linear function: I have to get better in the multiplication between each perceptron of the previous layer (i-1 layer) and the corresponding weight between the previous and current perceptron values and accumulate this result regarding the input that receives the current perceptron we are computing, so in the multiplication and accumulation part of the code.

I thought to optimize it in two different ways:

- Manual way: as I did in the first part of the second assignment, I could have implemented by myself the optimisation mechanism using parallelization with neon registers.
- Automatic way: setting in the compiler the optimization option and let the compiler do the parallelization by itself. Also, this point can be done in different ways and I tried both:
    - -o2 -ftree-vectorize: as I did in the second assignment which is the ready-to-use implementation of the usage of neon registers.

- -o3: which optimizes as the -o2 level (in our case is essential the use of loop unrolling, so the parallelization) and even simplifies the execution with two different optimizations in function calling.

I tried the optimization for all the digits but then I reported the results only for my three digits (0,5,3) which sometimes were also the bests with respect to all digits. During my tests I noticed something strange: if I execute the elf file different times, I get different results with the same number "digit 5" like for example 686us and in the next execution (very few seconds after) for example 32517 us. I thought that this can depend on how much the virtual machine is busy, I totally exclude that this depends on the different process running on the instance of the processor because it only had this neural network computation. I tried to make the computations simultaneously while a colleague was computing lonely and we get certain values, but after the log out of one of the two, the times were different.



```
Image loaded

Executing net...
Execution done cnn time took 686.00 us.

            TOP 0: [7] digit 7   ########################################### 89.4%
            TOP 1: [1] digit 1   ####### 6.7%
            TOP 2: [3] digit 3   ## 2.0%
            TOP 3: [9] digit 9   ## 1.5%
            TOP 4: [2] digit 2   # 0.3%




root@xilinx-zcu102-2018_1:~# ./dnn.elf
DATASIZE [B]: 2
MAXVALUE    : 32767
MINVALUE    : -32768
Initialization...
File ./weights/Gemm0_weights.bin exists
File ./weights/Gemm0_biases.bin exists
File ./weights/Gemm1_weights.bin exists
File ./weights/Gemm1_biases.bin exists
File ./weights/Gemm2_weights.bin exists
File ./weights/Gemm2_biases.bin exists
File ./weights/Gemm3_weights.bin exists
File ./weights/Gemm3_biases.bin exists
Init done

Loading image...
Loading txt file
Image loaded

Executing net...
Execution done cnn time took 32517.00 us.

            TOP 0: [7] digit 7   ########################################### 89.4%
            TOP 1: [1] digit 1   ####### 6.7%
            TOP 2: [3] digit 3   ## 2.0%
            TOP 3: [9] digit 9   ## 1.5%
            TOP 4: [2] digit 2   # 0.3%
```

Figure 10: here I show different times between one execution and the immediately next one. Both have same digit, same network parameters. When the first execution (with time 686 us) ended, I immediately started another execution (with time 32517 us)

I compared the results with a colleague from a different group with a total different network structure, I have one more layer and much more weights and bias. We had very different times, my little value was around 400us while the colleague value was around 100us. This is obvious because one more Gemm and ReLU layers to be executed need more time and even much more weights and biases to compute for each layer (like 4 or 2 times more for each layer) need much more time, so execution time increases.

After this I executed the network 100 times and saved the execution time of each iteration, computing max, min and average time, all of them expressed in us.

Table 1: This table shows execution time for all the 3 digits (0, 3 and 5) I had to consider. For each one I show times without optimization (o0 second column), with autovectorization (Autovect o2, third column) and with the highest optimization possible (o3, last column). For each optimization you find min, max and average time for 100 executions. All times are expressed in us.

	o0	Autovect o2	o3
0	Min: 26087.00 us Max: 152067.00 us Avg: 72763.00 us	Min: 459.00 us Max: 35456.00 us Avg: 3250.76 us	Min: 461.00 us Max: 20927.00 us Avg: 1284.65 us
3	Min: 28078.00 us Max: 212803.00 us Avg: 69154.24 us	Min: 464.00 us Max: 49461.00 us Avg: 3672.36 us	Min: 443.00 us Max: 52238.00 us Avg: 3846.17 us
5	Min: 26771.00 us Max: 195909.00 us Avg: 70465.54 us	Min: 420.00 us Max: 41923.00 us Avg: 2331.55 us	Min: 485.00 us Max: 47287.00 us Avg: 3012.16 us

Here below I also reported the trend for each one of my digits without optimization (blue line), with autovectorization and o2 (orange line) and with the highest optimization value I could set in Vivado SDK which is o3 (green line). In the y-axis you will see the time expressed in us while in the x-axis the number of the execution from 0 up to 99.

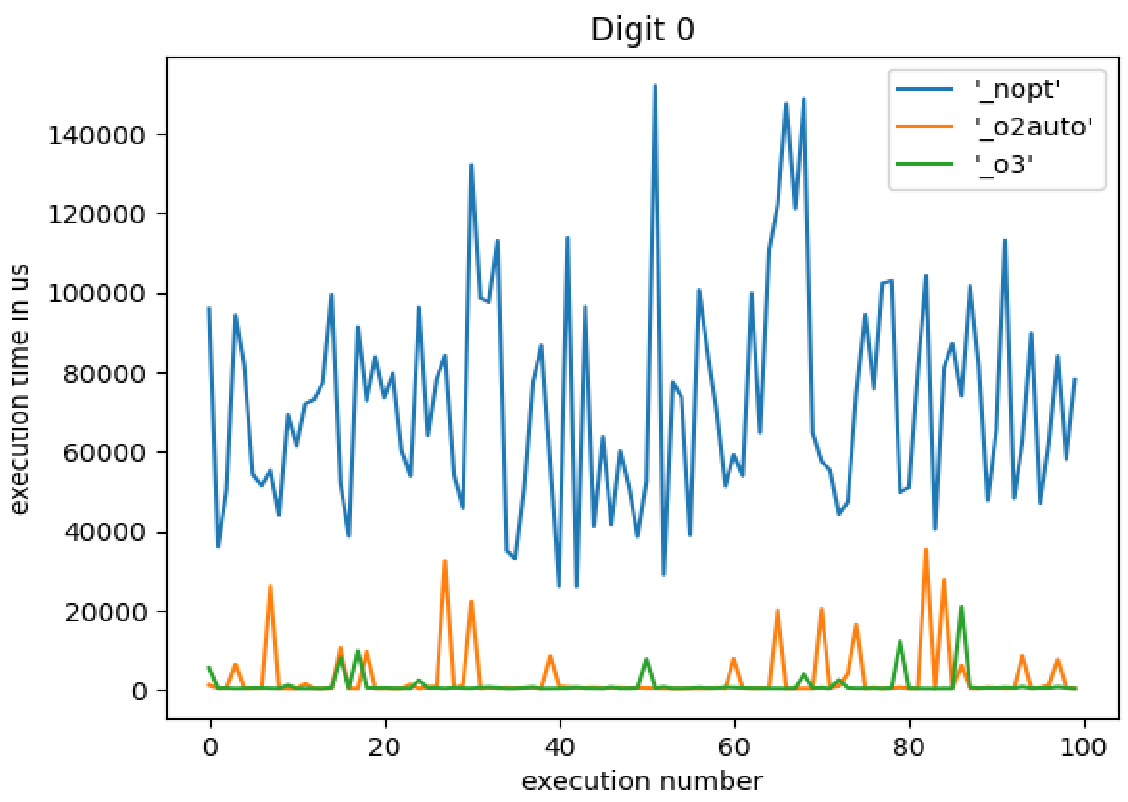


Figure 11: Graph explaining execution time for each optimization for each execution. Digit 0. Here is very clear the difference between 3 optimization levels and the best one is o3

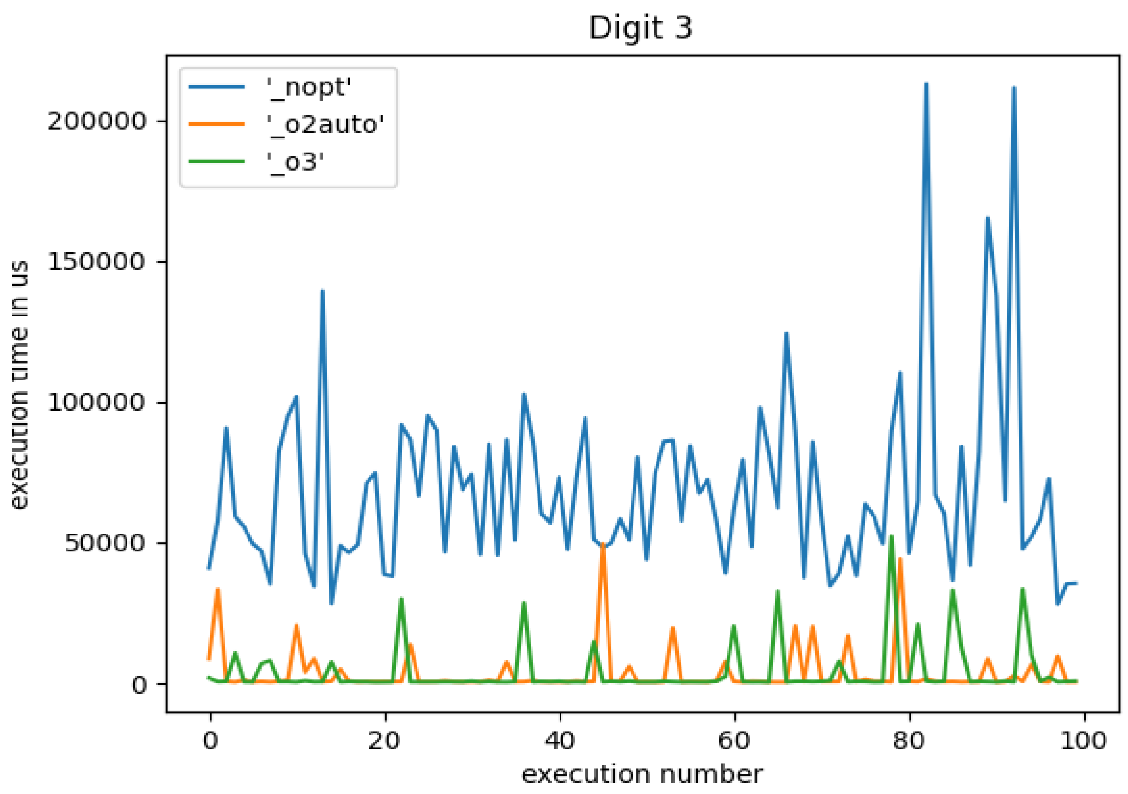


Figure 12: Graph containing execution time for each repetition. Digit 3. In contrast to performance's times in digit 0, here there is no such big difference between o2 and o3, as we can also see from average times in the Table 1

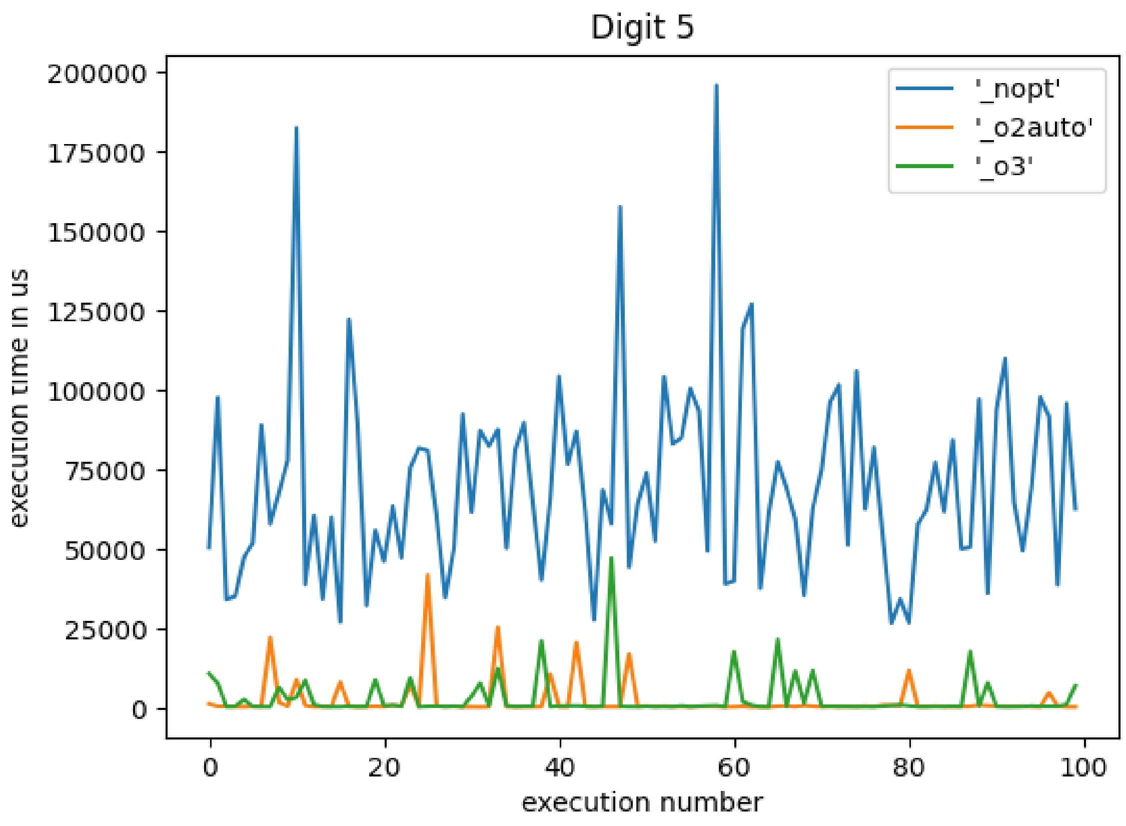


Figure 13: Graph showing execution time for each iteration. Digit 5. Here the difference between o2 and o3 is much higher than digit 3 but in contrast to digit 0, here the best performances are done with o2 autovectorization optimization as we can also see in Table 1 regarding average time, min and max time values.

Looking at these results I immediately noticed that there is a big difference between no optimization level and with the two optimization options. Regarding the o3 and o2 with autovectorization optimizations, I see that they are almost similar above all for digits 5 and 3 (pretty difficult to be rightly classified) while for digit 0 optimization level o3 leads very little execution times as expressed in the table above. Another interesting aspect is that digit 5 reaches the best times not only compared with itself but also to all optimization levels of all other digits, even if it had like the worst performance with respect to classification.