HeadHunter: Facial Features Recognition on PYNQ

Project Report for Xilinx Open Hardware 2018

Antonio Di Bello, Alberto Scolari, and Anna Maria Nestorov

Politecnico di Milano, Italy

{antonio.dibello,annamaria.nestorov}@mail.polimi.it, alberto.scolari@polimi.it

June 30, 2018

Supervisor: Prof. Marco Domenico Santambrogio, `marco.santambrogio@polimi.com`
University: Politecnico di Milano, Italy
Team number: 415
Link to YouTube video: `https://www.youtube.com/watch?v=BngGK_HhvtQ`
Link to Facebook page: `https://www.facebook.com/HeadHunterPYNQ`

Abstract

With embedded devices becoming ubiquitous in everyday life, a variety of applications arise from the possibility to recognize how many people are present in a scene. These applications span multiple contexts, from security to marketing, just to name a few, with more applications being possible as more details are captured (gender, age, facial expression, ...). While several techniques exist to detect human faces inside images, Neural Networks (NN) are showing up as a promising solution capable to detect multiple details (pose, landmark, gender, ...). However, adapting a NN model to embedded devices like the PYNQ platform is far from trivial, and requires to thoroughly explore many design decisions. To explore the potential of the PYNQ platform, our goal is to study how to perform face detection on this platform in a real-life-like scenario. This requires working with the whole software-hardware stack available on the platform, especially with the programmable logic, to leverage the integration capabilities of the PYNQ platform.

1 Introduction

Recent technological trends like Internet of Things (IoT) introduce new possibilities to collect diverse data and increase the demand for novel devices and applications. Sensors located in multiple environments allow gathering multiple data , which need to be integrated together to mine useful information from them. In this vast panorama, multiple business applications need information about people present in a space, possibly large, for a variety of goals like security, ads-campaigns, user-experience customization and so forth. This information should be extracted from an image representing the entire scene, which should be deeply analyzed to find relevant details like number of people in the scene and their gender, facial status and so on. The image is produced and first analyzed in the device "embedded" in the environment, which has usually limited capabilities and is power-constrained. Then, the image can either be processed locally or sent to a cloud-like facility for a more thorough analysis. The extraction of relevant people information from an image is very complex and requires many computations, which in turn leads to a high power consumption. When done inside the embedded device, this consumption may hinder the duration and the reliability of the device itself. However, the alternative of sending the data to a more powerful, unconstrained processing system is not always available, typically because the intermediate network is unavailable or too unreliable. Therefore, it should be possible to perform (most of) the computation locally, in order to guarantee the availability of the result.

The recognition of facial features can be done in multiple ways, and the scientific literature is rich of solutions for this task [9]. Works like [10] prefer performance over precision in recognizing faces, and perform well only with frontal views of faces, but fall short when the face is even partially rotated. A large body of recent literature prefer using NN-based classifiers for detection [7] as well as for feature recognition [4, 5], even if some work [11] used multiple tree-based classifiers for the same task. However, none of these (and of similar) works recognizes more than two features, and no work integrates face recognition with features recognition into a single solution. On the contrary, Hyperface [8] is a Convolutional NN (CNN) that integrates multiple recognizers into a single solution, which is able to extract the facial areas from the image as well as several interesting features (face landmark points, roll, pitch, yaw and gender).

Therefore, we adopted Hyperface architectures and ported it to our target embedded platform. We chose the PYNQ platform, which features a System on Chip with an ARM CPU and an FPGA. Since the NN computations for facial features recognition are often data-parallel at low granularity (e.g. the along weights of a NN), an FPGA is an ideal choice for this application, allowing to increase the energy efficiency of the calculation. The PYNQ platform, in particular, is particularly well equipped for this task, as it features a Xilinx SoC with an FPGA and a dedicated software stack that easies the prototyping of solutions for the embedded world, by employing a Python programming environment to gather data from sensors and peripherals and send tasks to the FPGA. Indeed, this work ports the Hyperface CNN architecture to the PYNQ, maintaining the high precision of this recognizer together with good performance, which is close to the software solution. Investigating speedup opportunities is thus left as future work, as it may require a tolerable loss in precision that highly depends on the application. Here, we demonstrate the effectiveness of our porting with a prototype that captures single frames from a USB-connected webcam and analyzes them, finally showing the image frame with the classification results (like fig. 2).

To show how this porting was carried out, section 2 explains the structure of the Hyperface CNN, while section 3 explains how it was ported to PYNQ, detailing the main challenges and their solutions. Then, section 4 discusses the results achieved, and section 5 draws overall conclusions on the porting experience, discussing possible future works and how this solution is publicly available for the Xilinx Open Hardware contest.

2 The Hyperface CNN and its applications

Hyperface is a CNN based on AlexNet [6], whose structure is visible in fig. 1 with the dimensions and the parameters of each layer. At first, the image is split into multiple candidate regions via Dlib [3], which identifies the most interesting regions to be given in input to the Hyperface CNN. Hyperface CNN borrows the base five convolutional layers from AlexNet, dropping the fully connected layers as they encode non-needed information for image classification. Instead, it adds two convolutional layers *Conv1a* and *Conv3a* to AlexNet's *Pool5* layer to obtain feature maps of the same dimensions, which can be concatenated into a single input that *Conv_all* reduce in size. Then, a fully connected layer (*FC_full*) transforms the feature maps into a single features vector, which is passed to the detectors of the specific features. Once the CNN computation is done, the result of the first featurization (face/non-face) is checked against a pre-defined threshold to be a promising candidate regions, or is dropped. Surviving candidate areas are further checked for containing partial faces (via the landmark points) and re-analyzed via the CNN. This whole process repeats three times, and at the end the surviving regions that overlap are merged together (via their landmark points) and the features are output.

Fig. 2 shows an example result of an image to which Hyperface was applied, with faces within squared boxes whose color depends on the gender, the facial feature of each person that are visible and the face orientation values (roll, pitch, yaw) on top of each surrounding box (in red color). It is also possible to see that faces without a sufficient number of features, either because cut out of the image or because largely covered by other objects in the scene, are not recognized.

Although these limitations, Hyperface is the state-of-art solution to recognize multiple facial features with a unique solution, which is desirable on an embedded system with limited resources. Furthermore the information it provides are valuable for a number of applications. For example, the gender information helps understanding the distribution of spectators to

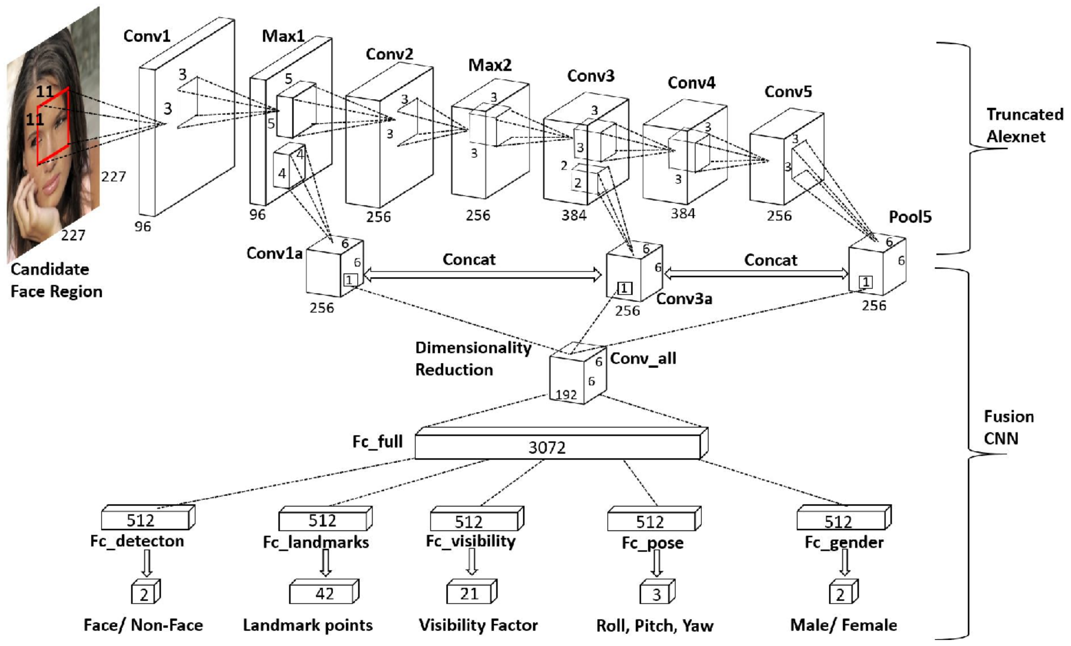

Figure 1: Structure of the Hyperface CNN, from [8]

an event, an information which in turn can be applied for a variety of goals like customer characterization, dedicated marketing, etc. The face landmark points can be used, e.g., to understand the emotional status of people, for example to measure the appreciation of an event. As a final example, the face orientation can be useful to track the response to visual stimuli like ads or warnings, checking whether people are attracted by them and what their appreciation is. For all these reasons, the richness of use cases these data can be used in motivates our choice of using Hyperface as the recognizer architecture for our implementation.

3 Implementation on PYNQ

To implement Hyperface on the PYNQ platform, we moved most of the logic into the FPGA. In particular, we ported to FPGA the convolutional (with ReLu) layers, the max plus normalization layers and the pooling layers, leaving the fully connected layers in software. These layers, indeed, are heavily bandwidth-bound, and their implementation with FPGA logic achieves little or no benefit with respect to the software implementation.

To fully implement Hyperface, we did not perform any quantization nor any precision reduction of the weights and biases, which are natively stored and used as single precision floating point data. Indeed, our goal is to fully implement Hyperface without any loss in the precision, in order to experiment with the full capabilities of the network and investigate its performance and its usage.

3.1 Kernels design

For each kernel, feature inputs and (where present) weights and biases are stored in the main memory, which is connected to the kernel via DMA logic. For the design of the layers we used Vivado High Level Synthesis, and explored the design space via iterating the design with this tool. Since the inputs are typically much larger than the on-chip memory of the FPGA, it is fundamental to properly size the windows for the computation of each layer in order to fully use the available resources and achieve the best attainable performance.

In the design of the kernels, two major factors drove the design choices: the size of the weights and dimensions to compute along. The size of weights determines how many of

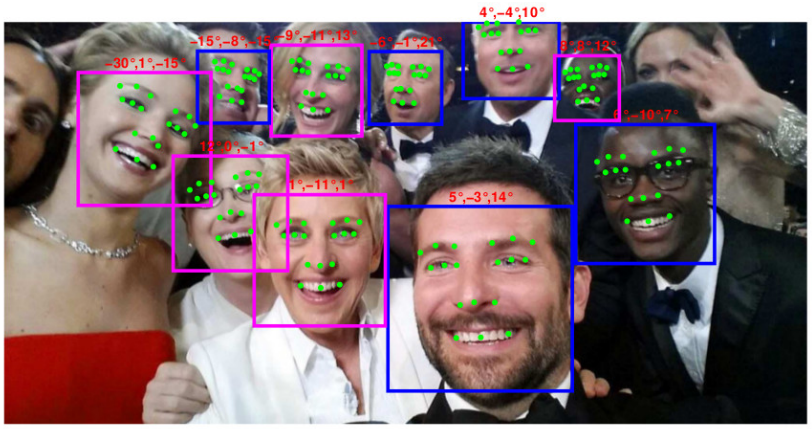

Figure 2: Example result of Hyperface recognition, from [8]

them can be cached on the on-chip memory, which ultimately dictates the number of the out channel that could be computed without needing new weights data transfer. Once the inputs are loaded, one has to consider the dimension along which each windowing computation of each kernel should be executed, as it dictates whether the result of the computation can be directly emitted or, more likely, has to be cached locally in order to be updated in future, once a new portion of input data has been fetched from memory. Therefore, each kernel was separately sized according to these criteria. An example is the Conv1 layer, whose pseudo-code structure is shown in listing 1.

Listing 1: Structure of kernel in Conv4

```
fm_height: for (int i = 0; i < DIMH-KH+1; i+=KS)
  fm_width: for (int j = 0; j < DIMW-KW+1; j+=KS)
    fm_out: for (int k = 0; k < FM_OUT; k++)
      # cache input portion from input stream into pixelInBuffer
      fm_in: for (int l = 0; l < FM_IN; l++)
        ...
        ker_height: for (int s = 0; s < KH; s++)
          ...
          ker_width: for (int t = 0; t < KW; t++)
            ...
            pixelOutBuffer += weights[k][l][s][t] *
                      pixelInBuffer[s][((j+t)*FM_IN)+l];
      # write pixelOutBuffer into output stream
```

Here, the six nested loops move along the three dimensions of the weights and the three dimensions of the convolution window, and the innermost body performs the multiply-and-accumulate (MAC) operation. To increase the performance of this code, inputs should be cached into registers or BRAMs, and loops from the innermost towards the outermost should be unrolled as long as registers for parallel access are available, accumulating the results of the unrolled loops via a reduction tree pattern, that is easily pipelined within the innermost remaining loop. Complete unrolling was possible for at most the two innermost loop, depending on the size of the convolution window. When this was not possible, BRAMs were to be used instead of registers (as the size of the window was too high) and the unrolling was limited to the innermost loop, with the input cache being properly partitioned to allow parallelism. In this case, partial results were accumulated into the same BRAMs, still allowing pipelining, or in intermediate registers to avoid tight dependencies between loops reading and writing from/to the same address.

3.2 System design

To communicate with the off-chip RAM memory, each kernel is connected to a DMA engine that is controlled via software. Then, the kernels extracts the data from the incoming stream and sends the values to the computational core. To pass parameters from the software to the kernel (such as the number of inputs, the type - weights or features - and so on), the kernels expose an AXI-Lite control interface that is connected to the Zynq processing system and mapped into physical memory, according to the usual practices of systems design with Vivado. Fig. 3 shows the block design of the Conv1 bitstream, with all the programmable logic components attached to the Zynq processing system and the DMA and control interfaces properly connected to RAM.

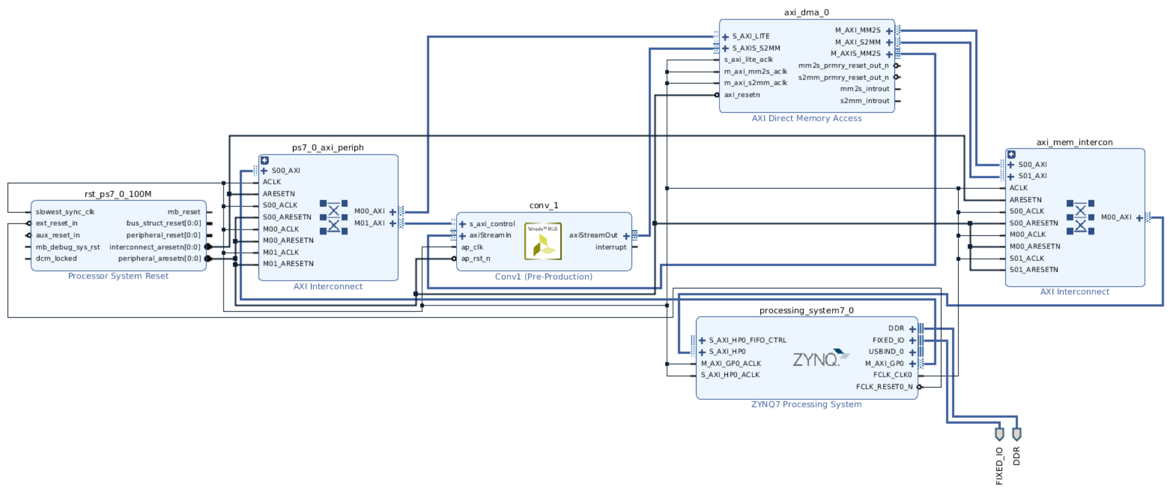

Figure 3: Block design of the system computing the Conv1 layer

3.3 Software design

The flow for recognizing faces and their features consists of multiple steps. As in Hyperface, the first step is the identification of *candidate regions*, which are regions where a face is likely present. This step is performed entirely in software via Dlib [3], which comprises very efficient and robust algorithms for this task.

Then, each candidate region is fed to the Hyperface CNN implemented in FPGA via a dedicated Python object, that returns the predictions. This object internally initializes the IPs from the bitstream files and allocates the buffer memory for communication with the FPGA logic. This memory, as from the best practices, is allocated through the Xilinx Contiguous Memory Area (CMA) interface provided by the PYNQ Python library. Since multiple bitstreams have to be flashed for the classification, reconfiguration overheads occur on the critical path of the execution, decreasing the performance. Therefore, we chose to batch multiple candidate regions and use the same instantiation of a kernel for a whole batch, thereby decreasing the number of reconfigurations by the batching factor, by default set to 8. Despite batching increases the latency of the computation, it is to be noted that the first step produces many candidate areas (several tens or hundreds of them) whose results are shown only at the end, and hence it is always possible to batch without delaying the final result, which is by default shown at the end of the computation.

3.4 Design Reuse

The hardware and software components of our design use the standard tools provided by Xilinx for Pynq, in particular the Python stack, Vivado HLS and Vivado. Therefore, these components can easily be reused in other designs and modified. In particular, HLS kernels, as they implement the CNN layers, follow common patterns, similarly to the layers themselves. The main customizations are the usage of windows buffers to cache weights in registers and the related pragmas to possibly unroll the execution (or pipeline it, when unrolling is not possible due to lack of resources). However, the structure of loops is quite general, and our IPs can confidently be adapted to different systems implementing the same functionalities, which are quite widespread in CNN-based solutions. Furthermore, they do not employ quantization techniques and they employ standard IEEE 754 since precision operations, thus they do not need any special re-training, input conversion or care during design. For these reasons, we believe others can benefit from our IPs for designing CNN prototypes, and we plan to release them as open source (with Apache V2 License [1]) via a code-sharing website; the webpage to retrieve the code will be advertised on the Facebook page of the project, shown in the front page of this report.

4 Results

Synthesis results highly depend on the IPs we wrote and on their parameters. Smaller designs such as Max5 use around 10% of the available resources, and can store the hot parameters in registers. Unlike them, other IPs make heavy use of resources like flip flops and LUTs, as well as using most of the BRAM available to store intermediate results and input weights, such as Conv4 and Conv5; in particular, we leave room for other components of the block design, especially the FIFOs of the DMA. This leads to multiple area/performance tradeoffs being applied to the IPs, with achieved target frequency of 100MHz.

Compared to the pure software implementation running on the ARM (which uses highly optimized libraries for the computation), out port is 60% slower on average, due to the lower frequencies the IPs run at and to the software overhead of reshaping feature inputs and weights to feed the FPGA kernel. Although the slowdown, our solution remains under a 10 minutes threshold to classify 100 proposed regions, which is a typical number from a monitoring camera capturing a scene with several people. This latency is compatible with our reference use cases, which have sampling times in the order of 10 to 15 minutes, and offloads most of the computation to the FPGA logic, with amore efficient and predictable behavior. Finally, our results are numerically very close to the software results, with observed differences on the sixth decimal digit of a small percentage of result values. These differences are due to the order of IEEE 754 operations (that are not associative) in some kernels, which is different from software because of the data reshaping. Nonetheless, these results do not impact the final classification result, because the variations occur in the last layers and neural networks are generally robust to approximations, which leaves large room for improvement as discussed in section 5.

5 Conclusions

With HeadHunter, we ported the Hyperface CNN to the PYNQ platform without changes to the CNN itself, showing that it is possible to retain the same classification precision within acceptable performance degradation for the use cases we consider. To this end, the PYNQ platform was instrumental in having easily accessible tools for prototyping, especially the support for the Python platform and the Python interfaces to handle and communicate with our kernels (flashing the FPGA, transmitting data through DMA, etc.). HLS tools were as well fundamental to quickly implement the IPs, allowing us to start from the C code and quickly explore the various design choices. This allowed the whole project be carried out in roughly 10 weeks/man by people with limited to no expertise of FPGA design.

Several future works span from this effort, depending on the target scenarios. The first possible future work is the investigation of the performance and porting to different devices, in case some scenario need the same quality of results but need, for example, higher performance. Interesting target platforms for this work are those base on the Zynq UltraScale+ MPSoC, which offer more hardware resources. Another valuable direction, for scenarios that allow renouncing to some precision in favour of performance, is the quantization of the CNN weights and inputs: this work can lead to noticeable speedups due to the higher computational intensity quantized inputs would allow, and has been demonstrated in past literature work [2]. Indeed, with quantized inputs floating point arithmetics would be replaced with fixed point arithmetics and the data types size would likely shrink as well, increasing the achievable parallelism and operators density within the logic.

References

[1] *Apache V2 license.* URL: `https://www.apache.org/licenses/LICENSE-2.0`.

[2] M. Blott et al. "Scaling Neural Network Performance through Customized Hardware Architectures on Reconfigurable Logic". In: *2017 IEEE International Conference on Computer Design (ICCD).* Nov. 2017, pp. 419–422. DOI: `10.1109/ICCD.2017.73`.

[3] *Dlib library.* URL: `http://dlib.net`.

[4] Georgia Gkioxari et al. "R-CNNs for Pose Estimation and Action Detection". In: *CoRR* abs/1406.5212 (2014). arXiv: `1406.5212`. URL: `http://arxiv.org/abs/1406.5212`.

[5] Bharath Hariharan et al. "Hypercolumns for Object Segmentation and Fine-grained Localization". In: *CoRR* abs/1411.5752 (2014). arXiv: `1411.5752`. URL: `http://arxiv.org/abs/1411.5752`.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: `http://dl.acm.org/citation.cfm?id=2999134.2999257`.

[7] Rajeev Ranjan, Vishal M. Patel, and Rama Chellappa. "A Deep Pyramid Deformable Part Model for Face Detection". In: *CoRR* abs/1508.04389 (2015). arXiv: `1508.04389`. URL: `http://arxiv.org/abs/1508.04389`.

[8] Rajeev Ranjan, Vishal M. Patel, and Rama Chellappa. "HyperFace: A Deep Multitask Learning Framework for Face Detection, Landmark Localization, Pose Estimation, and Gender Recognition". In: *CoRR* abs/1603.01249 (2016). arXiv: `1603.01249`. URL: `http://arxiv.org/abs/1603.01249`.

[9] Ashok Samal and Prasana A. Iyengar. "Automatic recognition and analysis of human faces and facial expressions: a survey". In: *Pattern Recognition* 25.1 (1992), pp. 65–77. ISSN: 0031-3203. DOI: `https://doi.org/10.1016/0031-3203(92)90007-6`. URL: `http://www.sciencedirect.com/science/article/pii/0031320392900076`.

[10] Paul Viola and Michael J. Jones. "Robust Real-Time Face Detection". In: *International Journal of Computer Vision* 57.2 (May 2004), pp. 137–154. ISSN: 1573-1405. DOI: `10.1023/B:VISI.0000013087.49260.fb`. URL: `https://doi.org/10.1023/B:VISI.0000013087.49260.fb`.

[11] X. Zhu and D. Ramanan. "Face detection, pose estimation, and landmark localization in the wild". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition.* June 2012, pp. 2879–2886. DOI: `10.1109/CVPR.2012.6248014`.