Neural Nets on FPGA

A machine vision algorithm applied on MNIST dataset using hls4ml library

Course of Electronics for Applied Physics

Giordano Calvanese

University of Bologna, department of Physics and Astronomy

September 20, 2019

Abstract

In this paper we describe a machine vision neural net algorithm implementation on FPGA. The algorithm is trained on handwritten digit dataset MNIST. For NN IP generation it's used a library called hls4ml which is a really powerful tool for fast implementation of NN on FPGA.

1 Introduction

hls4ml[1] is a library that acts as a bridge between machine learning world on cpu/gpu like Keras (Tensorflow) and the realm of VHSIC Hardware Description Language (VHDL) for designing FPGA's Neural Net Intellectual Property (NN IP) cores for fast inference. This bridge has its roots on High Level Systesis (HLS), a way of synthesizing hardware from a pseudo-C++ code. hls4ml in fact automatically writes for you the HLS code that corresponds to the specified NN, it needs a json file for the architecture and a hdf5 file for weights.

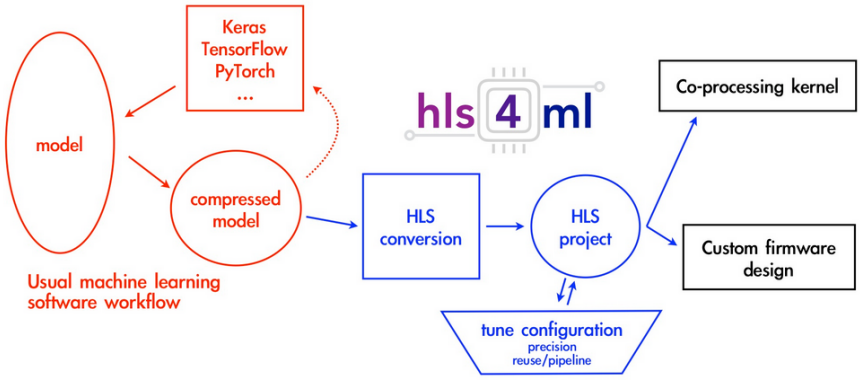
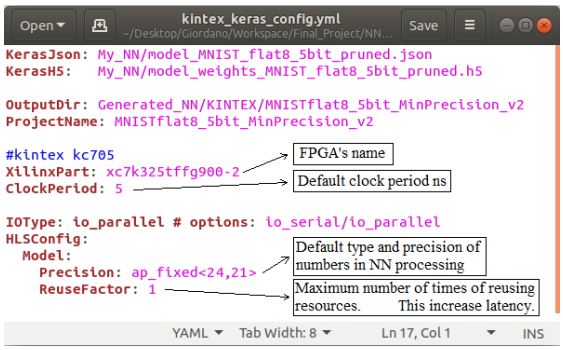


Figure 1: Hls4ml workflow. First phase: a) choose a model and train it on CPU (Keras in this case), b) iteratively, or just at the end, compress the model by pruning weights, c) export and save model's: architecture in json format and weights in hdf5 format. Second phase: a) generate HLS project using hls4ml, b) tune configuration of data type used by the model, c) IP core generation using hls4ml, which use in fact Vivado HLS. Third phase: Construct Vivado project for firmware design and import the NN IP core.

In order to generate HLS project, hls4ml need to compilate a yalm configuration file.



[1]Github repository: `https://github.com/hls-fpga-machine-learning/hls4ml`. Official site: `https://fastmachinelearning.org/hls4ml/`.

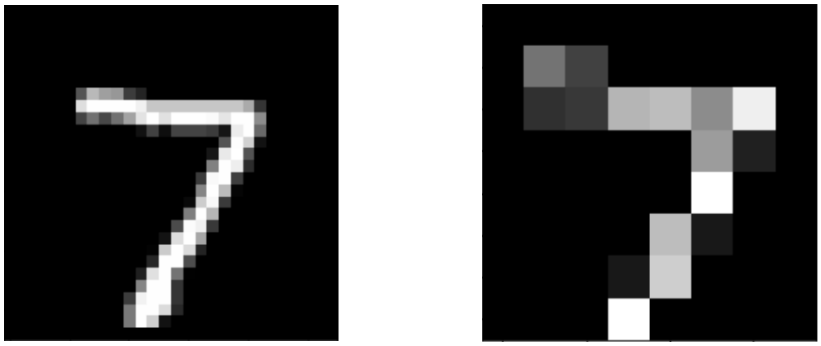
2 Choosing the model

Hls4ml is an ongoing work in progress and versions with new capabilities and bug resolution are near to come, nevertheless at the moment of writing 2D convolutional NN are not supported. In order to work with images one workaround is to flatten images into arrays and implement classical feed forward NN with dense layers.

Experiencing with capabilities of Vivado HLS to generate hls4ml's cores we find that nets with more than 100 nodes per layer causes a crash, probably both because of several vivado's critical warnings and excessive memory usage[2]. So we choose to work with images of size 8x8 or 64 input nodes.

An obvious choice is to work with MNIST dataset, which is a historic benchmark for machine vision tasks. MNIST dataset is a large ensemble of single handwritten digits images in 8-bit gray scale of size 28x28 pixels. MNIST dataset is composed of 60'000 images for training and 10'000 for test/validation.

In order to satisfy hls4ml limitations we decided to preprocess images by cropping the central portion of size 22x22 pixels and after that downsampling[3] for decrease resolution to 8x8 pixels.



(a) **(b)**

Figure 2: Comparison between original images and processed images used to train the model. **(a)** Original image taken from MNIST dataset: 28x28 pixels, 8-bit color depth. **(b)** Processed image: 8x8 pixels, 5-bit color depth.

The choice of transforming the color depth of pixels from 8-bit to 5-bit is simply for reasons of hardware resources usage: on one hand for keeping the NN IP core as small as possible[4] and on the other hand simply because

[2]We observed that the crash sometimes matched a complete RAM saturation. In our experiments we used an 8GB RAM.

[3]python: *scipy.ndimage.zoom()*

[4]Of course this color depth has proven, in training phase, the highest accuracy on test set but not that much higher respect to the 4-bit one. In other words, increasing in color depth at constant model causes on increase in test accuracy but with a 5-bit color depth we can reasonably say that we have almost reached the maximum possible accuracy.

these images will be printed on a screen using the VGA port present on FPGA's board which have 6-bit for green and 5-bit for red and blue.

After some experimentation with different architectures we find that the two hidden layer NN in **Figure 3** shows an excellent accuracy on both training set and test set[5], proving its capabilities of generalize well on new data.

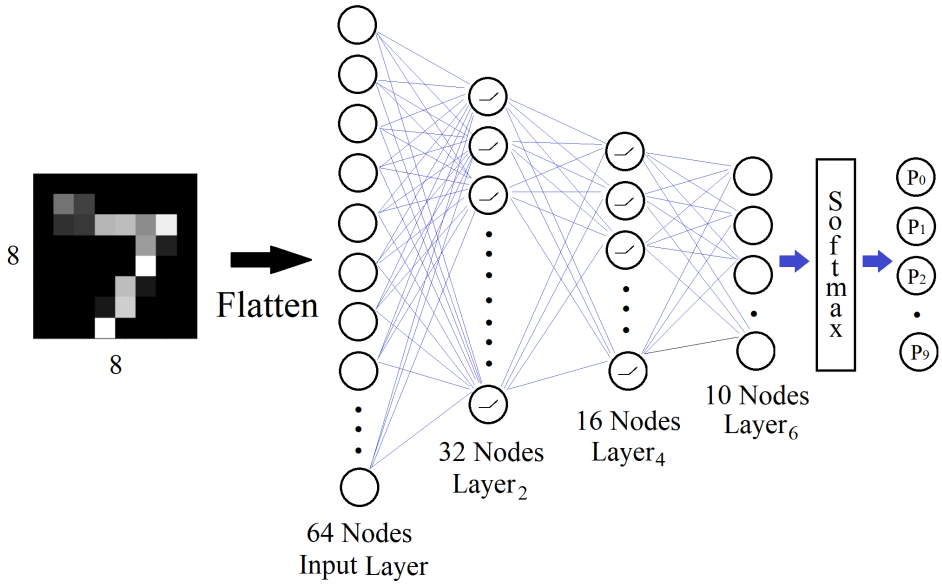


Figure 3: Representation of model's architecture. Symbols inside nodes stands for activation function, relu in this case.

3 Training and compression

After a training phase of 15 epochs[6] the accuracy of the model respect to training data has already reached the maximum of almost 94%.

Now that the model is trained we implement the compression. Hls4ml in fact has the important feature of write code for HLS that ignores and not implement unnecessary operations, like multiplications per zero, with net effect of reducing model's hardware resource request. To take advantage of this feature we have done the so called *pruning*, i.e. artificially setting to zero weights smaller than, in absolute sense, a certain cutoff.

[5]Actually it shows almost the same accuracy on both sets, which means that the net has learned the maximum amount of information without overfit training data.

[6]One epoch occurs when all example of training set are presented at least one time to the net for training purpose.

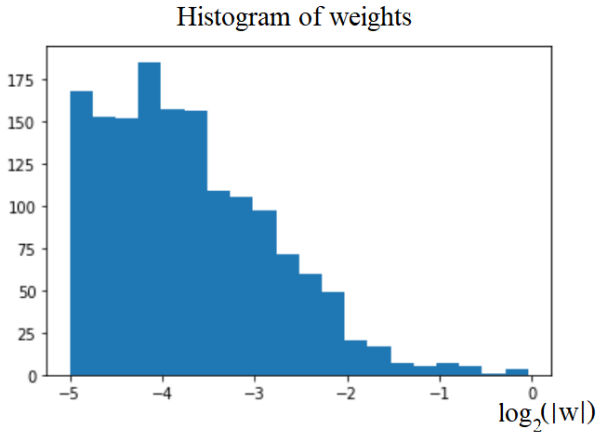


Figure 4: Histogram of absolute values of non-null weights of the trained model after pruning. It's clearly visible that the cutoff used is equal to 2^{-5}. Furthermore we can notice that the shape of this histogram resemble an half gauss bell and this is compatible with the fact that weights of the pruned model are almost a half in number than the weights of the complete one (see table in **Figure 5**).

	# Parameters	
	Complete model	Pruned model
Layer_2	2'048 + 32	979 + 30
Layer_4	512 + 16	363 + 15
Layer_6	170 + 10	130 + 10
TOTAL	**2'778**	**1'527**

Figure 5: Overview table of compression. Complete list of parameter for every layer (weights plus biases) pre and post pruning.

In next table we show a summary of performances. As you can see accuracies doesn't suffer so much from pruning.

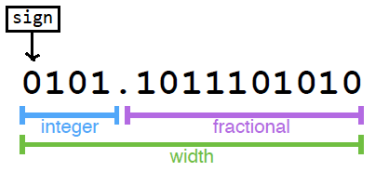
	Training accuracy %	**Test Accuracy %**
Complete model	0.9389	0.9471
Pruned model	0.9392	0.9423

Now we save and export the model in order to generate HLS project through hls4ml. We don't worry about "*Precision*" parameter in configuration file because data type and precision will be a significant part of data type tuning in next chapter.

4 Implementation on FPGA

Once we have HLS project of our NN, it's time to fine tuning the data type in order to fit our objective, consume less resource as possible and most important of all avoid overflow.

The most widely used data type in HLS is **ap_fixed<width, integer_size>** (arbitrary precision fixed point numbers). First of all, let's focus our attention on weights and biases (generically called w). We have:



$$0.03125 = 2^{-5} \leq \mid w \mid \leq 0.97244$$

ap_fixed<14,4>

Therefore we need just 2 bit as integer size I_s (one for sign and one for 0.) and at least 5 bit as fractional size F_s. If we choose 5 bit as fractional size we badly quantise our parameter's space, in fact we would than have a maximum relative error of 1. In order to better represent our space of parameters we choose a maximum relative error $\max\{\Delta_{rel}\}$ of $2^{-3} = 0.125$ or in other words 8 bit as fractional size.

$$F_s = -\log_2\left(\min\{\mid w \mid\}\max\{\Delta_{rel}\}\right) = -\log_2\left(2^{-5}2^{-3}\right) = 8$$

So for weights and biases should be enough **ap_fixed<10, 2>**.

Predicted classes probabilities[7] are, as always, less than or equal to 1. Therefore 2 bit as I_s are enough and we choose for the final layer **ap_fixed<16, 2>**.

For input we choose the data type **ap_int<6>**[8] because we have to take into account also sign.

In order to fine tuning data types for hidden layers we have to calculate the maximum value that each node can take; this is necessary to prevent overflow. For this purpose we introduce a new quantity for the i-th layer A_i which is the maximum value that nodes of that layer can takes. A_i is simply defined as[9] the product of the maximum weight for the $A_j \cdot \#node_j$ of the incoming layer.

- Layer$_2$: $A_2 = \max\{\mid w \mid\} \cdot A_{Inputs} \cdot \#node_{Inputs} < 1 \cdot 2^5 \cdot 2^6 = 2^{11}$

$$\Rightarrow I_{s,2} = 1 + \log_2(A_2) = 12$$

[7] Our NN returns probabilities due to the final Softmax operation.

[8] At the time of writing we discovered that we'd rather use **ap_uint<5>**.

[9] Because we are using relu as activation function, A_i for relu layers has the same value as the incoming layer.

- Relu_3: $A_3 = A_2 \qquad \Rightarrow \qquad I_{s,3} = I_{s,2}$
- Layer_4: $A_4 = \max\{|\ w\ |\} \cdot A_3 \cdot \#node_3 < 1 \cdot 2^{11} \cdot 2^5 = 2^{16}$

$$\Rightarrow I_{s,4} = 1 + \log_2(A_4) = 17$$

- Relu_5: $A_5 = A_4 \qquad \Rightarrow \qquad I_{s,5} = I_{s,4}$
- Layer_6: $A_6 = \max\{|\ w\ |\} \cdot A_5 \cdot \#node_5 < 1 \cdot 2^{16} \cdot 2^4 = 2^{20}$

$$\Rightarrow I_{s,4} = 1 + \log_2(A_5) = 21$$

We choose to assign as fractional size the value of 3 for all hidden layer: $F_{s,i} = 3$ for $i = 2, 3, 4, 5, 6$.

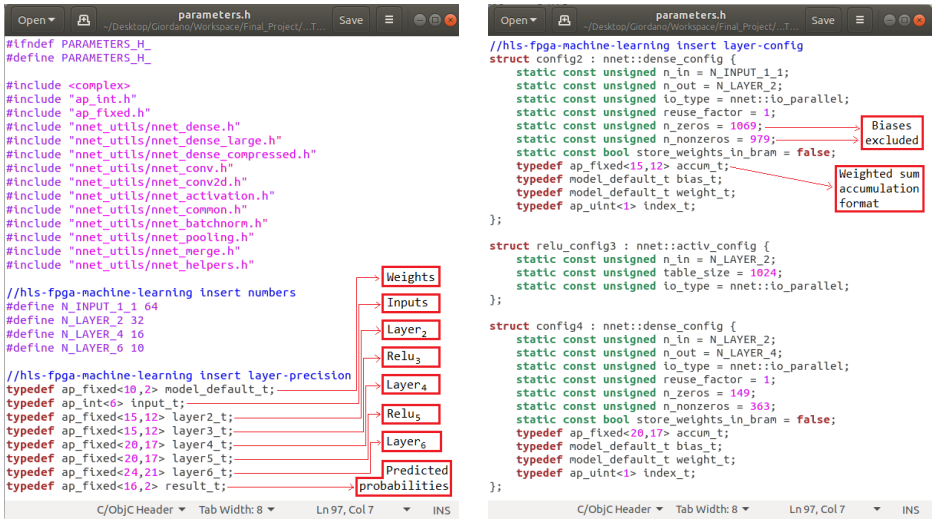


Figure 6: In order to fine tuning the NN we have to actively modify the file "OutputDir/firmware/*parameters.h*".

In the next table we show how data type fine tuning change demand for resources.

NN IP core version	LUTs	FFs	BRAMs	DSPs
Not-tuned[10]	58'109	58'628	25	512
Tuned	34'766	19'163	25	513

[10]The not-tuned version is simply obtained setting, in configuration file, the maximum required precision for weights and hidden layers: **ap_fixed<29,21>**; which corresponds to $I_s = 21$ and $F_s = 8$. Data type of inputs and predicted probabilities are tuned as usual.

Lab implementation

The FPGA's availability of our lab constrained us to use two FPGAs: a *Kintex kc705* for NN IP core processing and a *Zybo z701* for displaying input images on a screen through VGA port.

We writed into firmware of both FPGAs 100 images taken from MNIST test set. First 10 images corresponds to digits 0,1,..., 9; while other images were randomly choosen. Accuracy[11] of our pruned model on this set of images is happen to be 97%. This result is probably super-human, in fact it's really difficult for our eyes to recognize digits on some of these low-resolution images.

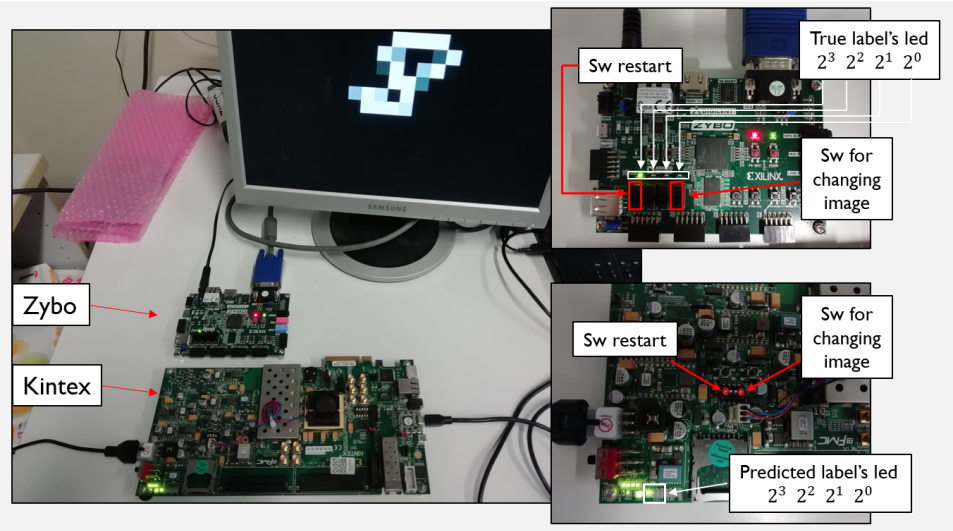


Figure 7: Implementation settings in laboratory. Switches control the changing in input images. Leds on Zybo prints out true label and leds on Kintex the predicted ones.

[11] This calculation was done using Keras.

5 Conclusion

In conclusion we evaluate the computational time needed to the NN in order to process an input image. This inference time on FPGA is than compared to the inference time required on CPU.

CPU[12]	FPGA[13]
1'982'269 ns	125 ns

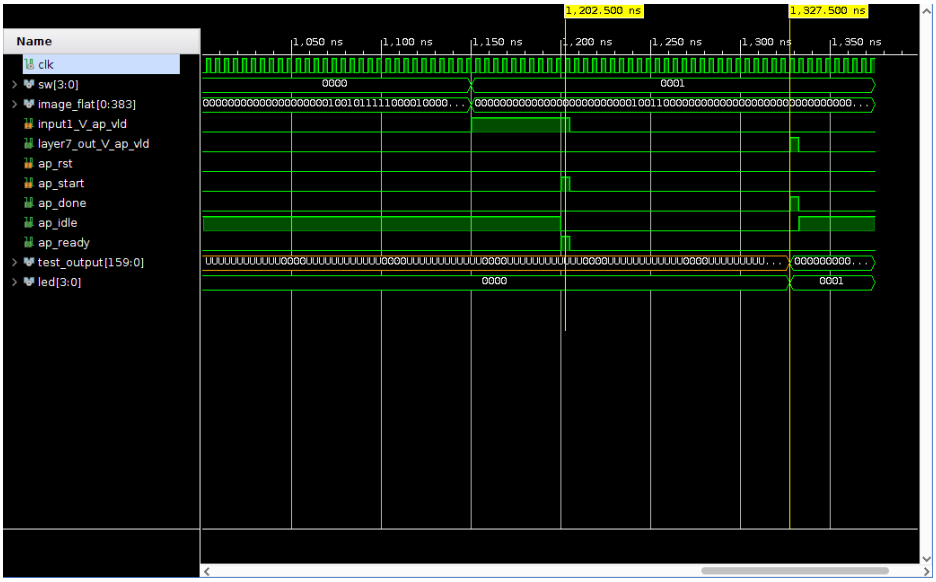


Figure 8: Vivado simulation for measuring the inference time or *latency* of the NN IP core. In this implementation version switches sw[:] controls the input image given to the NN: sw = 0 gives an image of a zero, sw = 1 gives an image of a one and so on. The NN IP core is initially in idle mode and in fact it's waiting for a valid input to process. The inference process is triggered by ap_start signal in concomitance of the input1_V_ap_vld signal which is the signal that validates the input (image_flat). After 25 clock cycles the NN IP core returns a valid output in concomitance of the ap_done signal. The predicted class probabilities are than processed and the class label predicted is visible through led signal.

One may ask if it is necessary to wait until the NN IP core has finished to process an input image before you can give to it another one to process. We have found that this is not the case as you can see in the next figure.

[12]This is relative to a Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz, running Keras routines.

[13]This is relative to a Kintex kc705 FPGA which has a 5 ns clock, in fact more general this NN's elaboration takes 25 clock cycles.

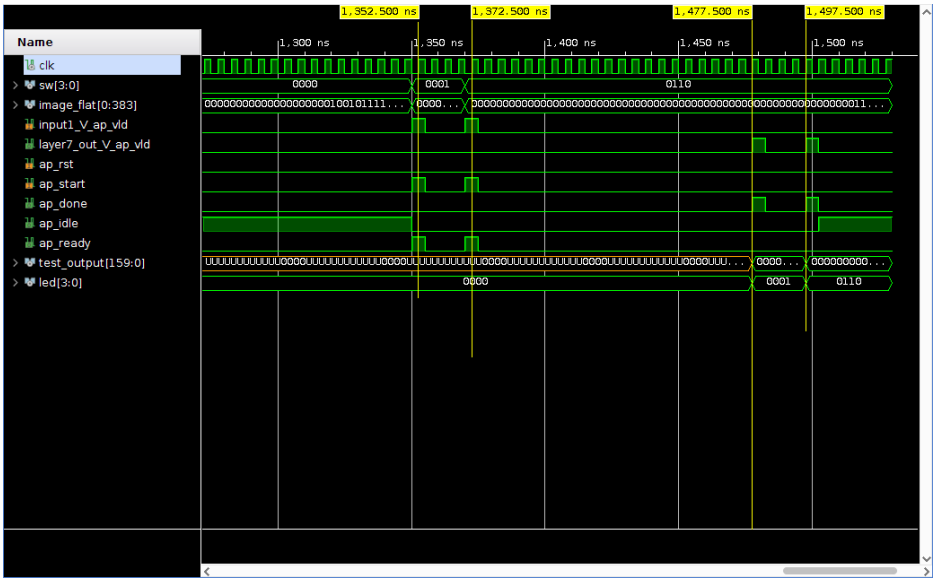


Figure 9: Vivado simulation that prove the NN IP core capability of pipelining inference. Two valid input images are fed into the NN with a time delay smaller than latency and the NN IP core is still able to process both.

So, latency of the NN IP core is not a limiting speed factor and therefore this is a really interesting feature in order to apply this kind of hardware for triggering event mechanism such as particle recognition in particle physics.

We finally want to point out that "*ReuseFactor*" parameter in hls4ml's configuration file has no effect, in term of resource usage, when applied to data type fine tuned NN IP cores. Benefits from this parameter instead can be seen when applied to not-tuned NN IP cores, but this will necessary cause an increase in latency.

References

[1] J. Duarte et al., "*Fast inference of deep neural networks in FPGAs for particle physics*", JINST 13 P07027 (2018), arXiv:1804.06913.