An FPGA Implementation of a Deep Autoencoder using hls4ml package

ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA
DEPARTMENT OF PHYSICS AND ASTRONOMY
APPLIED ELECTRONICS COURSE

Lorenzo Valente

April 23, 2022

Abstract

In this report is presented an implementation of a *Deep Autoencoder* architecture trained on the MNIST database in FPGA (Field Programmable Gate Array), focusing on machine vision tasks for the data reconstruction and classification in the latent dimension. To implement machine learning (ML) models in FPGAs, a companion compiler based on High-Level Synthesis (HLS) called hls4ml is used. In addition, an optimization using both compression and quantization of Neural Neutwork (NN) is perferomed in order to obtain sensible reduction in model size, latency and energy consuption.

Contents

1 Introduction

The hls4ml package was developed by members of the High Energy Physics (HEP) community to translate ML algorithms into HLS code, enabling firmware development times to be drastically reduced. In this project development, hls4ml is used as the tool to perform this transformation on a trained Deep Autoencoder model. A schematic workflow is shown in figure 1.

The goal of the hls4ml package is to empower a HEP physicist to accelerate ML algorithms use FPGAs, thanks to their tools for the conversion. Indeed, hls4ml translates Python objects into HLS, and its synthesis automatic workflow, allowing fast deployment times also for those who know how to write software or are not yet experts on FPGAs.

Hardware used for real-time inference usually has limited computational capacity due to size constraints, and incorporating resource intensive models without a loss in performance poses a challenge. One efficient way to reduce a models size, is through post-training quantization, where model parameters are translated into equivalent lower precision parameters. However, this processes, by definition, lossy and sacrifices model performance. Therefore, solutions to do *quantization-aware training* have been suggested. In chapter 5, results of quantized compared with non-quantized model is presented. This relatively new technique implemented by hls4ml enable to incorporate resource intensive models without a loss in performance, providing efficient interference. In this case, a fixed numerical representation is adopted for the whole model and the model training is performed enforcing this constraint during weight optimization.

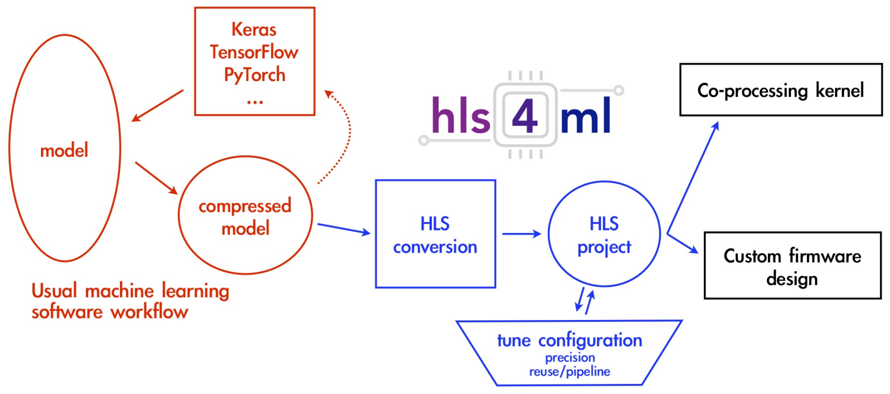

Figure 1: A typical workflow to translate a model into an FPGA implementation using hls4ml.

2 Data preparation

The first important step before model creation is data preparation.

The Vivado HLS capabilities enable to generate hls4ml's cores with at most 1024 parameters calculation, i.e. 10 bit, for each neural network's layer. Considering this constraint, as well as the consistency of the model's output, a reduction in the input image resolution is performed on the original datasets.

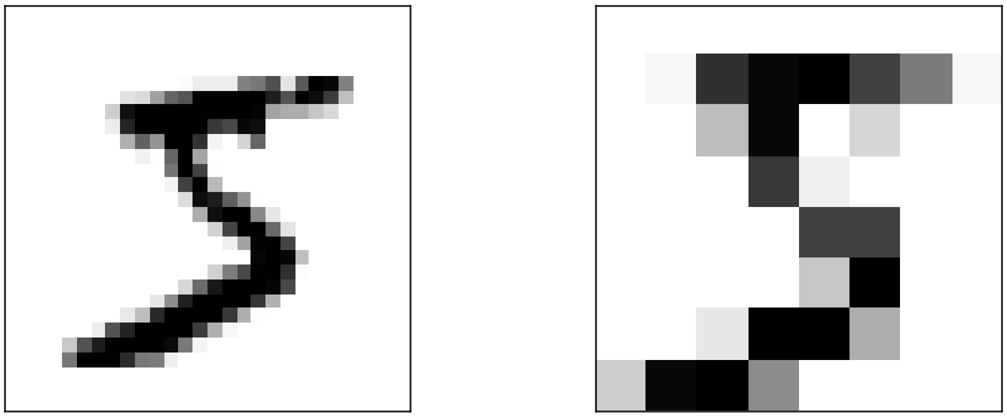

Figure 2: On the *left* an image from the original dataset. On the *right* the modified dataset after the preprocessing.

The MNIST database is chosen as input for our model. The original database contains handwritten digits, from 0 to 9. MNIST has been divided into two datasets: the training set has 60,000 samples and the test set has 10,000. Each image contains a single handwritten digit, each one with 28x28 pixels ranging from 0 to 255, i.e. 8-bit grayscale value.

In particular, the preprocessing that we performed on the original MNIST database consists of:

- **cropping the central portion** from size 28x28 to 22x22 pixels;
- **downsampling** for reducing the resolution from 22x22 to 8x8 pixels;
- **transforming of the color depth** from 8-bit to 5-bit.

The last point is chosen simply for reasons of hardware capability. The images in figure 2 are shown as an example, the first image from the original training dataset and the same figure after the preprocessing.

3 Choosing the Model

Autoencoding is a data compression algorithm, where the compression (*encoding*) and decompression (*decoding*) functions are:

- **data-specific**, therefore they will only be able to compress data similar to what they have been trained on;
- **lossy**, which means that the decompressed outputs will be degraded compared to the original inputs;
- **learned automatically from data examples**, it means that it is easy to train specialised instances of the algorithm, that will perform well on a specific type of input.

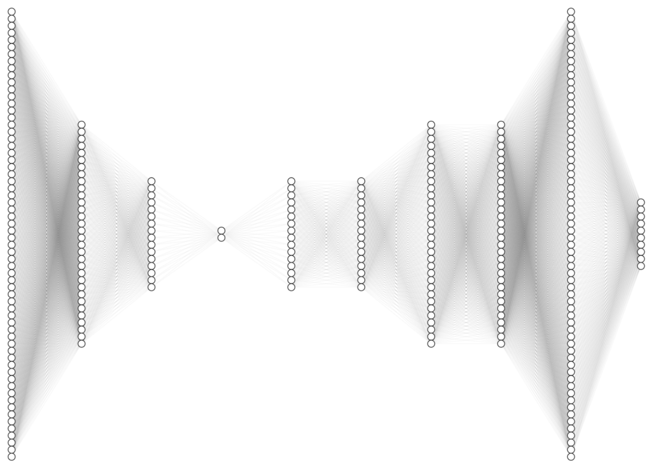

Figure 3: total model

The design of a **deep fully-connected autoencoder** implemented uses an architecture which imposes a *bottle-neck* on the network. It forces a compressed knowledge representation of the input data. In this implementation, it is used compression in *two-dimensional* latent space. Compression and reconstruction are extremely demanding if there is no structure in the data, i.e. no correlation between input features. However, if some sort of structure exists in the data, it can be learned and applied when forcing the input through the bottleneck. The model has been considered both with and without the classifier for the decompression.

3.1 Autoencoder for the Reconstruction

After some experimentation with different node sizes, considering the consistency of the algorithm's output as well, we find an optimal structure for each layer as depicted in figure 4.

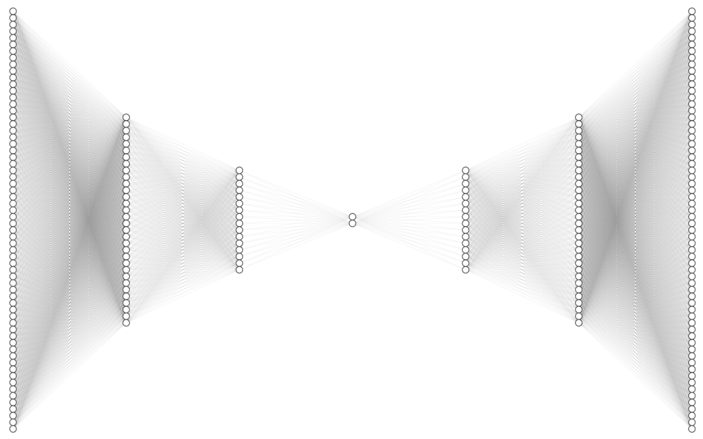

Figure 4: Implemented autoencoder model. The **encoder** (*left part*) takes 64 nodes in input layer, 32 and 16 nodes for hidden layers respectively ending up into a compressed two-dimensional latent space. The **decoder** (*right part*) takes as input the two-dimensional latent space and decompress the information in complete ascending analogy as the previous encoder structure, ending up into a 64 nodes layer as output.

The loss function used to train an under complete autoencoder is called *reconstruction loss*, as it is a check of how well the image has been reconstructed from the input. Although the reconstruction loss can be anything depending on the input and output, we will use an L1 loss to depict the term represented by the equation (1):

$$L(r, \hat{r}) = |r - \hat{r}|, \tag{1}$$

where r is the ground truth and $\hat{r}$ represents the predicted output.

All the layer nodes are activated by the *ReLU* function, except for the last output layer which is activated by the *sigmoid* function.

3.2 Classifier

To evaluate the correct reconstruction performed by the autoencoder's reconstruction, a classifier has been implemented.

The neural networks architecture, shown in figure 5, is attached to the encoder part in the two-dimensional latent space and it works in parallel with the decoder for the decompression. In contrast with the previous architecture, we are now focusing on the classification problem. Therefore, for the output layer, the activation function *softmax* is used, so we can train the model exploiting the knowledge on the labels from the input datasets.

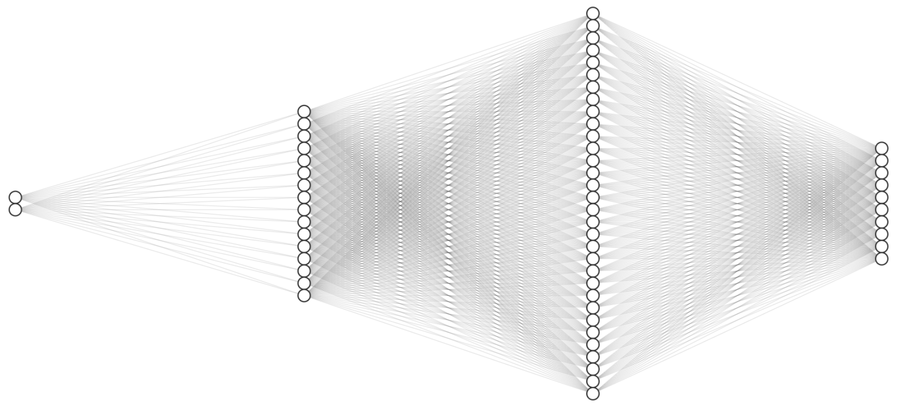

Figure 5: Implemented classifier model. It takes as input the compressed space by encoder and decompress the information with two hidden layers of 16 and 32 nodes respectively. Ending up with a 10 nodes output layer, one for each digit classification.

4 Neural Network optimization via Compression

The final hardware platform where the inference computation will run must be considered when building a neural network model. To minimize the resource utilization when a neural network is targetted to FPGA, aimed at reducing the number of parameters and operations involved in the computation, by removing connections and thus parameters.

This process is commonly called **weight pruning**, i.e. the elimination of unnecessary values in the weight tensor, by practically setting the parameters' values to zero, which will be translated in cutting connections between nodes during the synthesis of the HLS design. The pruning is done during the training process to allow the NN to adapt to the changes.

The TensorFlow Sparsity Pruning API was selected to perform this optimization. It uses an algorithm designed to iteratively remove connections, based on their magnitude during training. Therefore, a final target sparsity, i.e. target percentage of weights equal to zero is specified, along with a schedule to perform the pruning. For our experiment, we start pruning at step 2000, stop at step 10000 and do it every 100 steps. As training proceeds, the pruning routing is scheduled to execute, removing the weights with the lowest magnitude, until the current sparsity target is reached.

4.1 Neural Network model performance on CPU

Pruning a model can harm accuracy. To obtain the best accuracy for our model, we explored the trade-off between accuracy, speed and model size. Therefore, we avoid pruning the first layer and the critical layers for compression and decompression of the information. We proceeded to prune only the hidden layer with 16 nodes in both compression and decompression parts, we reached a sparsity of 85%. In the figure **??** are shown the distributions of the weights through the layers. It can be noticed that in the histograms for the *second*, *fourth* and *fifth* layer we observe large peak in the bin containing '0'.

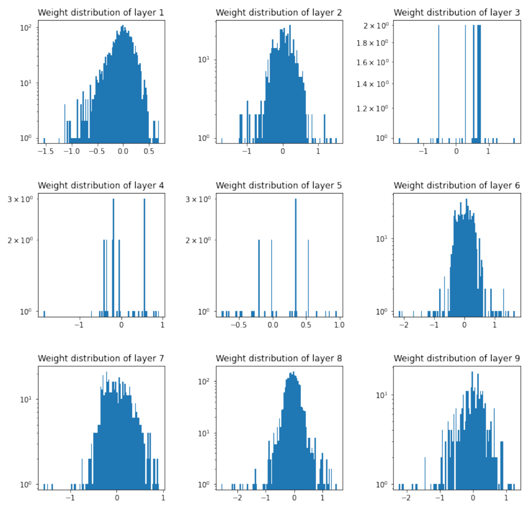

Figure 6: Weights distributions of the *complete* model with classifier.

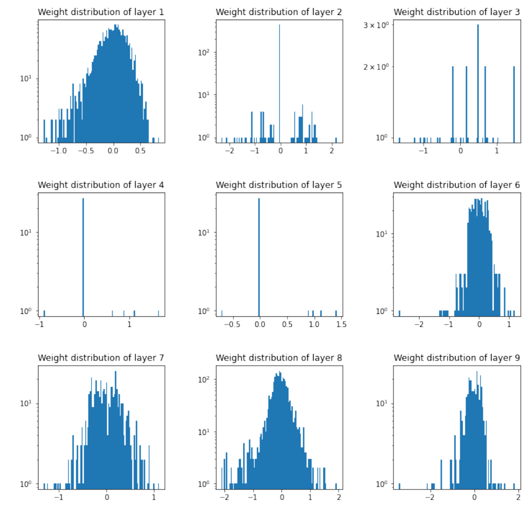

Figure 7: Weights distributions of the *pruned* model with classifier.

Summary of the pruning technique in the tab 1. As we can see, we reached a reduction in the total number of parameters of almost 0.93%.

	Complete model	Pruned model
2nd layer	512 + 16	102 + 16
4th layer	32 + 16	6 + 13
5th layer	32 + 16	6 + 15
Total parameters	**6268**	**5805**

Table 1: Results of the pruning technique for the model with classifier. There is a complete list of parameters before and after pruning in the format *weight + bias*.

Plotted below are model score losses and classifier accuracies. The model has been considered both with without the classifier for the decompression. Model score losses, computed according to the equation 1, approach zero at high epochs as expected. As it can be noticed, bumps appear near the eighth epoch, meaning that the algorithm for iteratively removing network connection start work.

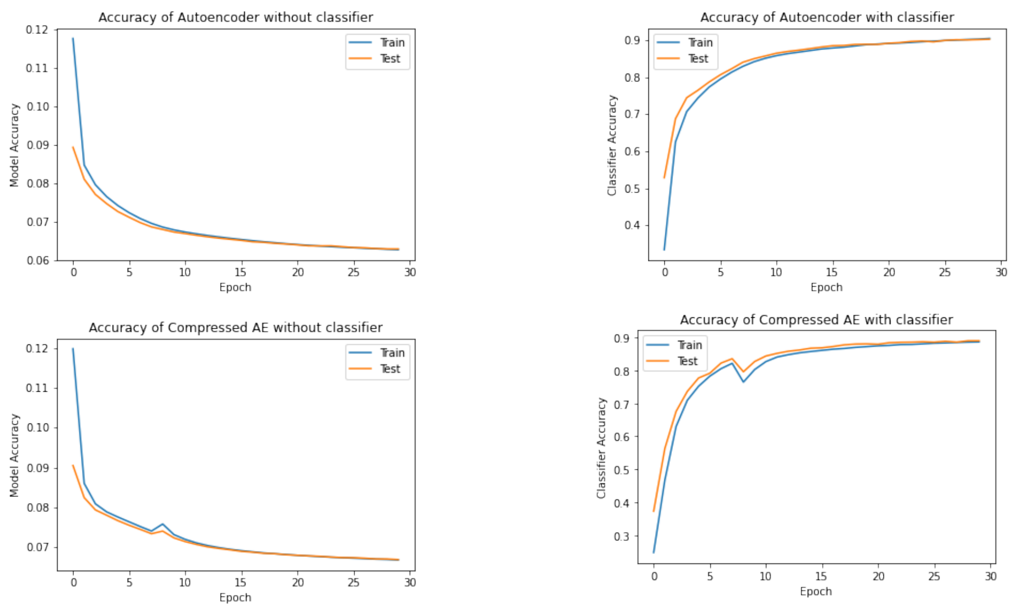

Figure 8: On the *left* an image from the original dataset. On the *right* the modified dataset after the preprocessing.

In the following table 2, we show a summary of performances after a training phase of 30 epochs. Accuracies and losses do not suffer so much from the pruning procedure.

	Complete model	Pruned model
Training loss	0.0626	0.0668
Testing loss	0.0628	0.0669
Training accuracy	0.9041	0.8863
Testing accuracy	0.9020	0.8897

Table 2: Final results of the performances for pruned and complete model with classifier.

Plotted below is the distribution of labelled data in its two-latent dimension space. In the latent space, it can be noticed a linear distribution of images. Each of these coloured clusters is a type of digit. Close clusters are structurally similar digits, i.e. digits that share information in the latent space. This particular linear distribution describes the fact that we have two dimensions to express a handwritten digit. Then it could happen for certain digits that the height increases and the width increase as well, creating a linear shape in the latent space, as displayed.

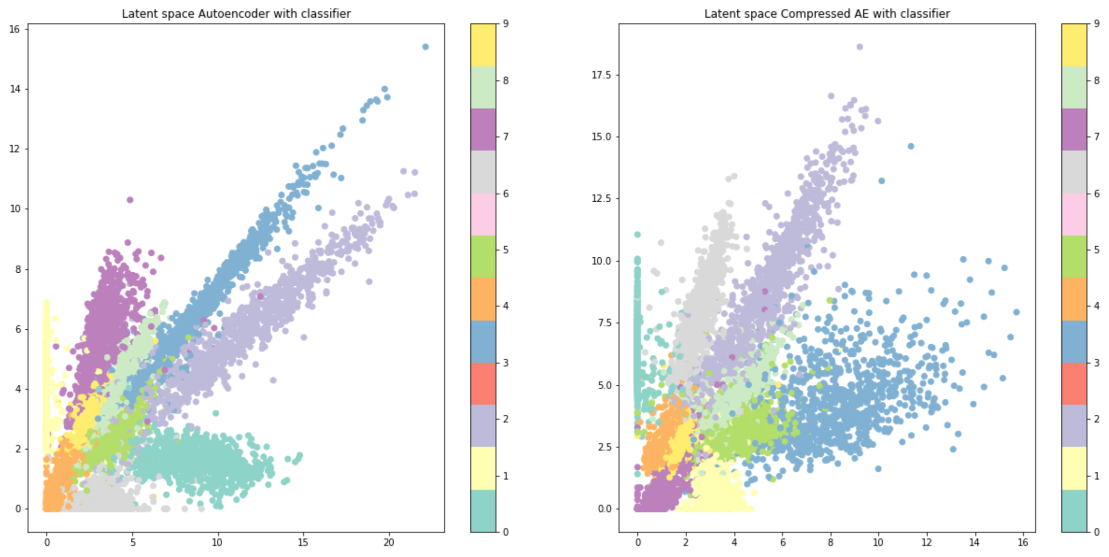

Figure 9: Digit distributions in the latent space for the model with classifier.

The last two images of this section represent the original training dataset on the first row and for the second row, the reconstructed images after both the complete and pruned model are applied.

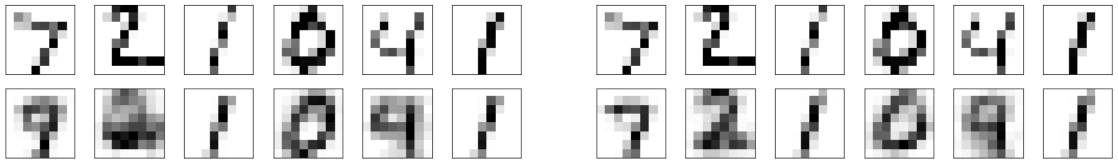

Figure 10: Reconstructed images.

The models trained in the previous section were saved into HDF5 files, containing:

- the architecture of the model, to be reproduced when it is needed to be used with other datasets;
- the model weights;
- the training configuration (losses, optimizers);
- the state of the optimizer, allowing to resume training exactly where it was left.

5 Neural Network optimization via Quantization

Quantization, which means conversion of the arithmetic used within the NN from high-precision floating-points to normalized low-precision integers (fixed-point), is an essential step for efficient deployment.

Fixed-point numbers consists of two parts, integer and fractional as shown in figure. Compared to floating-point, fixed-point representation maintains the decimal point within a fixed position, allowing for more straightforward arithmetic operations.

To simplify the procedure of quantizing Keras models, the QKeras library [ref], written on top of Keras, has been developed by a collaboration between Google and CERN. QKeras enable independent quantization of trainable parameters of layers and intermediate tensors. It provides a rich set of quantizers to choose from, enabling mantissa quantization, exponent quantization, binary and ternary quantizers.

5.1 Quantization of NN and performance on CPU

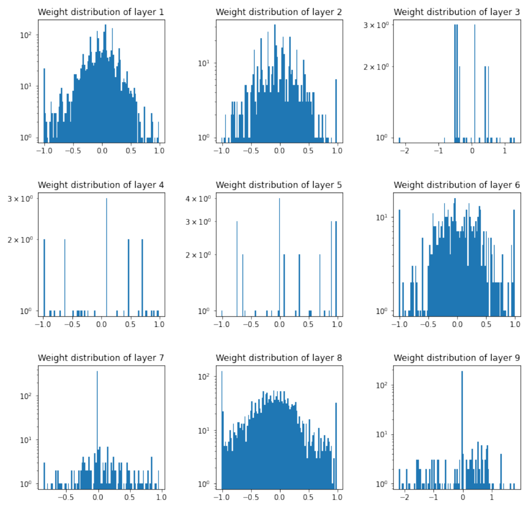

Figure 11: Weights distributions of the *quantized complete* model with classifier.

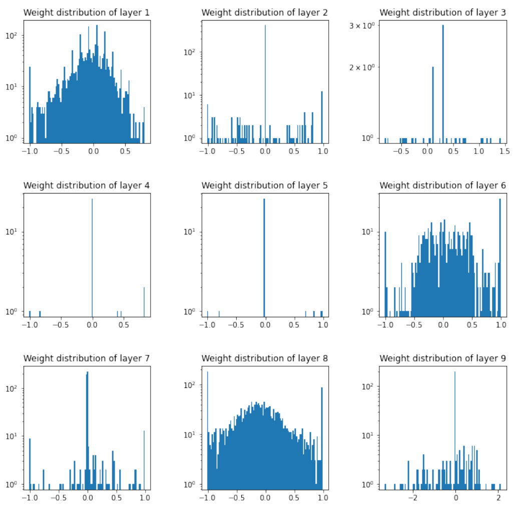

Figure 12: Weights distributions of the *quantized pruned* model with classifier.

6 Implementation of NN on a FPGA

7 Conclusions