An FPGA Implementation of an Autoencoder using hls4ml package

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

Lorenzo Valente[*]

May 5, 2022

Abstract

In this report it 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. Furthermore, an optimization using both compression and quantization of Neural Networks is performed to obtain sensible reduction in model size, latency and energy consumption.

[*]E-mail: lorenzo.valente3@studio.unibo.it

Contents

1 Introduction

Members of the High Energy Physics community have developed hls4ml to translate ML algorithms into HLS code, enabling firmware development times to be drastically reduced [2]. The tool used for this project development is the hls4ml[1] library based on a trained Deep Autoencoder model. Figure 1 shows a schematic workflow.

The goal of the hls4ml package is to empower a HEP physicist to accelerate ML algorithms using 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

[1]https://github.com/fastmachinelearning/hls4ml

challenge. One way to reduce a model's size is through post-training quantization, where model parameters are translated into lower precision equivalents. This process, by definition, is lossy and sacrifices model performance. Therefore, solutions to do *quantization-aware training* have been suggested.

The results of the quantized compared with the Non-quantized model are presented here. Through the use of this relatively new technique, hls4ml can integrate resource-intensive models without sacrificing performance and resulting in efficient interference. In this case, a fixed numerical representation is employed for the entire model, and the model is trained with this constraint during weight optimization.

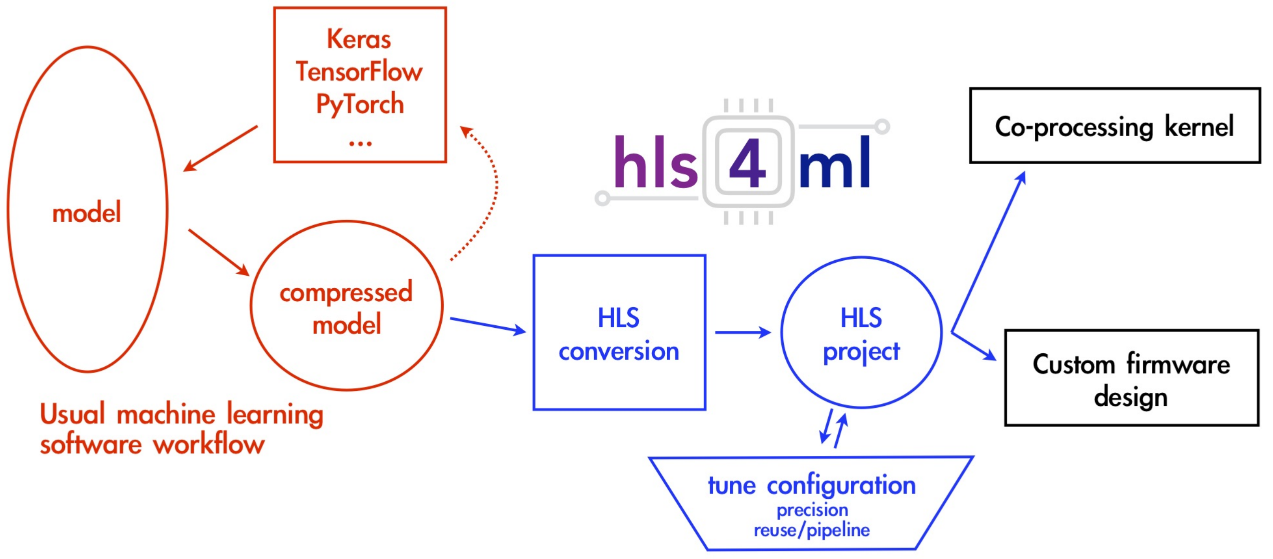


Figure 1: A typical workflow to translate a model into an FPGA implementation using hls4ml.

2 Data pre-processing

Data preparation is the first crucial step before model creation.

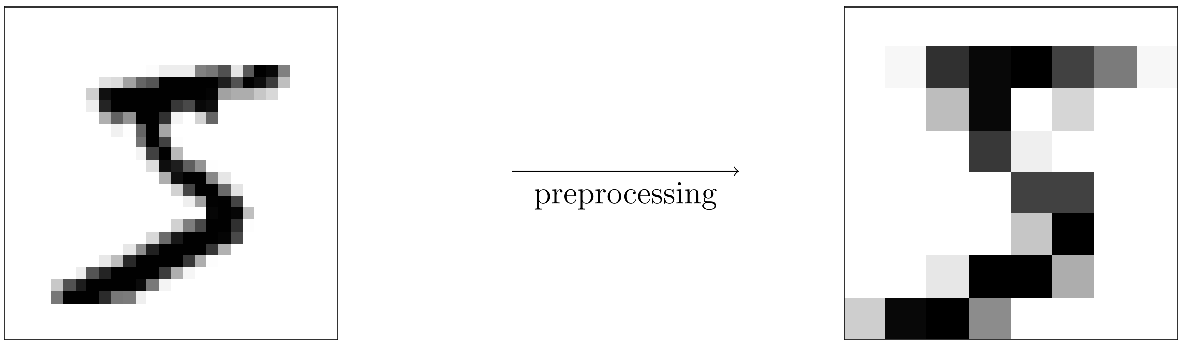
The Vivado HLS capabilities enable the generation of hls4ml's cores with at most 1024 parameters calculation, i.e. 10 bit, for each neural network's layer. Due to this constraint and the consistency of the model's output, the original datasets are reduced in resolution.

The MNIST database is chosen as input for our model. The original database contains handwritten digits, from 0 to 9. MNIST is comparted into two datasets: the training set has 60,000 samples and the test one set has 10,000. Each image contains a single handwritten digit 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. Below are shown an example of the images before and after the preprocessing.



(a) (b)

Figure 2: The figure 2a is an image from the original test dataset. The figure 2b is the modified image after the preprocessing.

3 Choosing the Model

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

- **data-specific**, so they will only be able to compress data that is similar to what they 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.

The design of a **deep fully-connected autoencoder** implemented uses an architecture which imposes a *bottleneck* 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

An autoencoder [4] is a neural network trained to copy its input to its output. Typically, they are restricted in a way that allows them to copy only approximately and to copy only input that resembles the training data. By prioritizing which parts of the input to copy, the model learns useful properties of the data.

Traditionally, autoencoders were used to reduce dimensionality or to learn features. Recently, theoretical connections between autoencoders and latent variable models have brought autoencoders to the forefront of generative modelling.

After some experimentation with different node sizes, considering the consistency of the algorithm's output as well, we found an optimal structure for each layer as depicted in figure 3 [6]. We have reached a compression in a *two-dimensional latent space*, i.e. a compression of our initial data of $\sim 97\%$.

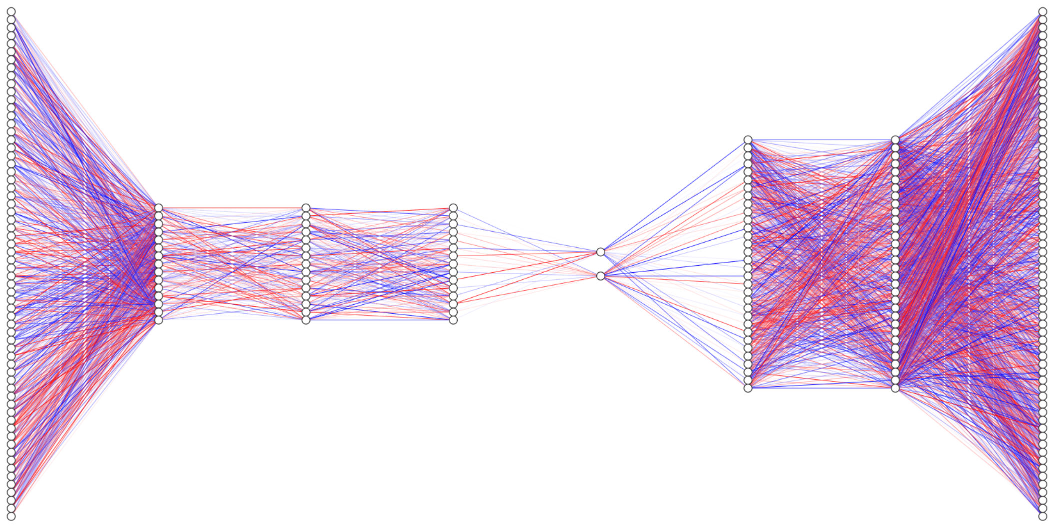


Figure 3: Implemented autoencoder model. The **encoder** takes 64 nodes in input layer, i.e. one for each pixel in input data; ending up into a compressed latent space. The **decoder** takes as input the two-dimensional latent space and decompress the information in complete ascending analogy as the previous encoder structure, ending up into a 64 nodes layer as output for reconstructing the image. Edge widths are proportional to edge weights. The color are proportional to edge weights: blue for negative edges and red for positive ones.

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.

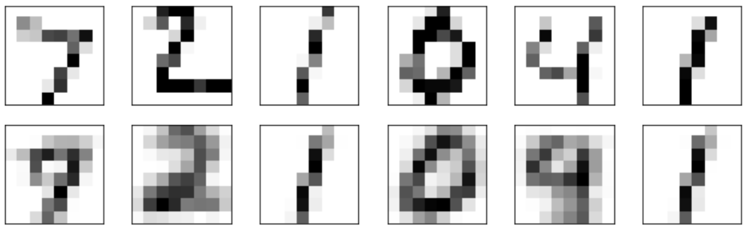


Figure 4: First row is the initial Autoencoder's input. The second row is the Autoencoder's output.

3.2 Supervised Classifier

A classifier has been implemented in order to evaluate the correctness of the autoencoder's reconstruction.

The neural networks architecture, shown in figure 5, is attached to the encoder 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 by exploiting the knowledge of the dataset labels.

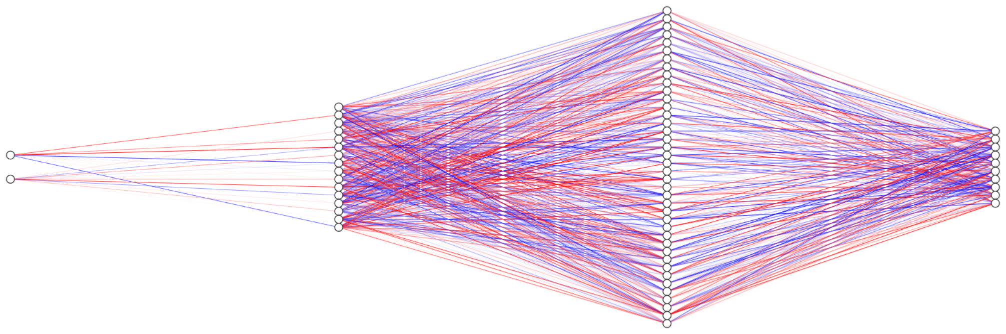


Figure 5: Implemented classifier. Its output ends up with a 10 nodes layer, one for each digit to be classified.

4 Neural Network optimization via Compression

It is fundamental to consider the hardware platform where the inference computation will run when developing 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 means 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[2] performed this optimization. The algorithm works by removing 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. As training proceeds, the pruning routing is scheduled to execute, removing the weights with the lowest magnitude, until the current sparsity target is reached.

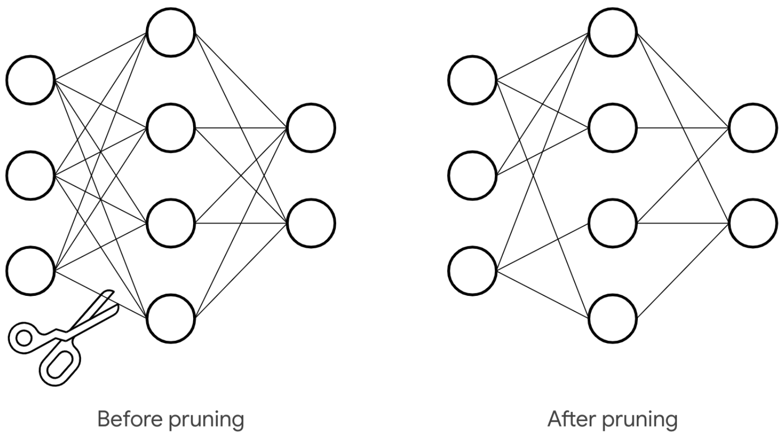
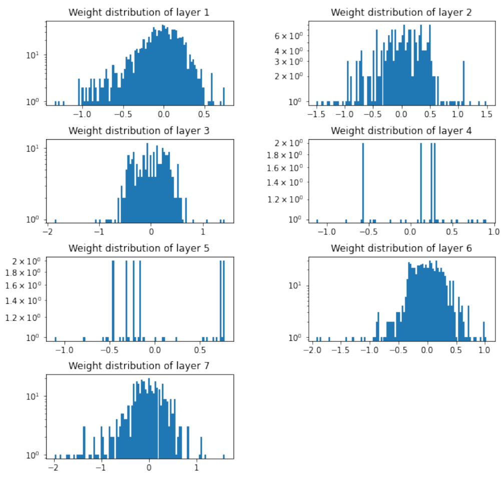


Figure 6: Graphical representation of the weight pruning optimization.

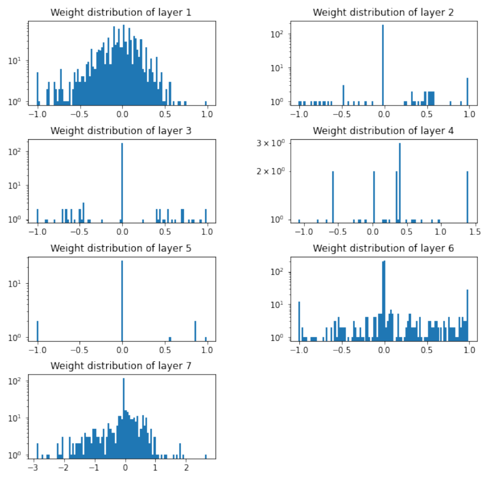
4.1 Compression's performance on CPU

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 critical layers for compression and decompression of the information. We proceeded to prune only the first two hidden layers in both compression and decompression parts. In the figure 7 are shown the weight distributions through the layers. It can be noticed in figure 7b a large peaks in the bin containing '0', meaning that we reach a target sparsity of almost 85% for the *second*, *third* and *fifth* layers, as we imposed.

[2] https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html



(a)



(b)

Figure 7: In 7a the weight distributions of the *complete* model, in 7b the *pruned* ones.

LayerName	**Complete model**	**Pruned model**
second pruned	225 + 15	34 + 14
third pruned	225 + 15	34 + 15
fifth pruned	32 + 16	5 + 10
Total parameters	**6385**	**5969**

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 size + bias size*.

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 6.5%.

Plotted below are model score losses and classifier accuracies. The model has been considered both with and 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 connections starts to work.

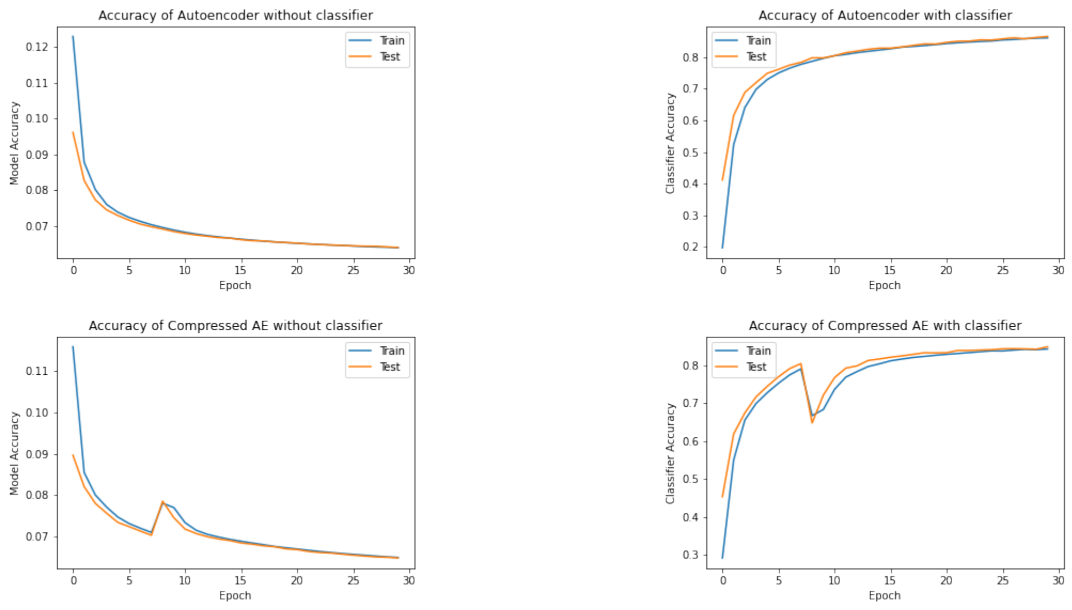


Figure 8: On the first row we can appreciate the complete Autoencoder's accuracy both with and without the classifier. On the second row we see the training behaviour after the compressed model.

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	0.0637	0.0649
Testing loss	0.0637	0.0648
Training accuracy	0.8821	0.8663
Testing accuracy	0.8800	0.8697

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

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 consist of two parts, integer and fractional as shown in figure 9. Compared to floating-point, fixed-point representation maintains the decimal point within a fixed position, allowing for a more straightforward arithmetic operations [3].

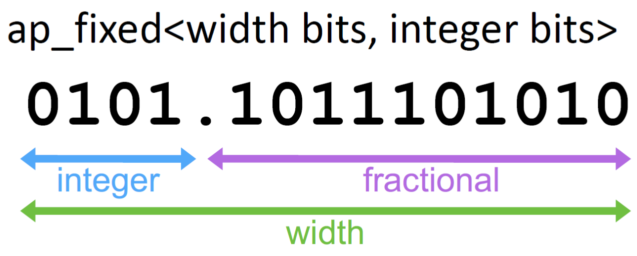


Figure 9: Fixed-point number representation. It will be the C type associated to input, output and parameters of the NN model by the hls4ml library.

To simplify the procedure of quantizing Keras[3] models, the QKeras[4] library, 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, and binary and ternary quantizers.

5.1 Quantization's performance on CPU

The last step before converting into an HLS project our NN, it's fine-tuning the data type. To reach our objective, consume fewer resources as possible and most important, avoid overflow. The default format used by Keras' layer is **ap_fixed<16,6>**. Instead, to obtain the best optimization for our model, we explored the trade-off between accuracy, speed, model size and type, similarly as we saw in 4.1 for each layer.

In tab 3 is reported the complete list of parameters used for the quantization.

For input we choose the data type **ap_ufixed<6,0>**, since after the data pre-processing, we obtained 5-bit defined for each image in the database. We can notice that it uses instead 6 bits in the first layer input for the type, since QKeras does not count the sign-bit when we specify the number of bits, so the type that gets used needs 1 more. This definition reflects also in all the other layer precision for weight and biases but is not required by the activation ones. We can also notice that for the reconstruction procedure it is useful to have an additional bit

[3]https://keras.io/
[4]https://github.com/google/qkeras

	Precision		
LayerName	**weight**	**bias**	**activation**
encoder input	ap_ufixed<6,0>	—	—
first	ap_fixed<5,1>	ap_fixed<5,1>	ap_ufixed<5,0>
second pruned	ap_fixed<5,1>	ap_fixed<5,1>	ap_ufixed<5,0>
third pruned	ap_fixed<5,1>	ap_fixed<5,1>	ap_ufixed<5,0>
encoder output	ap_fixed<16,6>	ap_fixed<16,6>	ap_fixed<16,6>
fifth pruned	ap_fixed<6,1>	ap_fixed<6,1>	ap_ufixed<6,0>
sixth	ap_fixed<6,1>	ap_fixed<6,1>	ap_ufixed<6,0>
classifier output	ap_fixed<16,7>	ap_fixed<16,7>	ap_fixed<16,6>

Table 3: Per-layer quantization for the baseline model (encoder + classifier). The same precision is used for weights and biases.

to have higher precision. For the *critical layers*, i.e. encoder output and classifier output, we set a different data type just for a bit of extra performance in the encoding information and for the classification.

The weight distributions for each layer is shown below in 10. As we can notice for the second, third and fifth layers it remains the compression as we saw previously in 4.1.

Plotted below are model score losses and classifier accuracies. The model has been considered both with and without the classifier for the decompression. Model score losses converge also in this case, as expected. As it can be noticed, in figure 11 the bumps are more evident with respect the previous situation in 8, meaning that the algorithm for iteratively removing network connections have more constraint in the data type, and requires 10 additional epochs to reach a stable a minimum in the *variance-bias trade off* [5].

In the table 4, we show a summary of performances after training. Accuracies and losses do not suffer so much from the pruning procedure.

	Quantized model
Training loss	0.5125
Testing loss	0.5198
Training accuracy	0.8683
Testing accuracy	0.8716

Table 4: Final results of the performances for the quantized model.

The models trained in the previous sections 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;

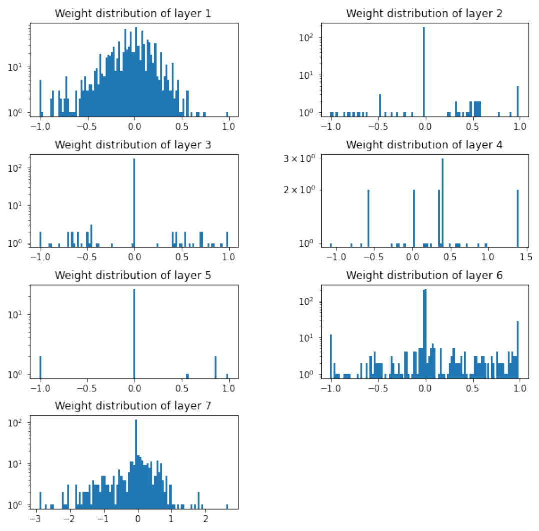
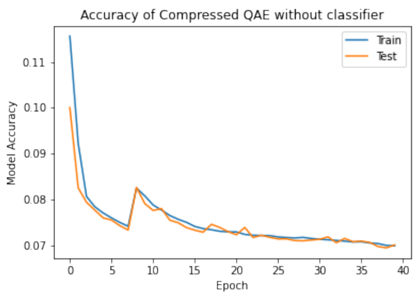


Figure 10: Weight distributions for each layer after the quantization-aware-training procedure.



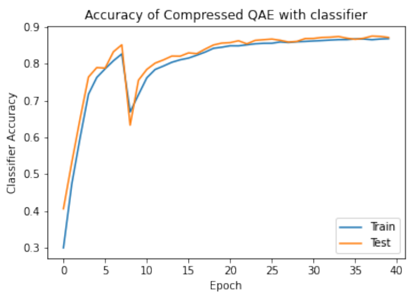


Figure 11: Comparing with the Non-quantized model training phase in figure 8, we notice how the constraint on type representation makes the training arduous with high fluctuation while reaching a convergence, so more epochs are required now.

- the training configuration (losses, optimizers);
- the state of the optimizer, allowing to resume training exactly where it was left.

6 Implementation of a Neural Network on a Field Programmable Gate Array

This section will present the main subject of the work described in this report: the implementation on an FPGA compression and classification inference performed by an Autoencoder using data coming from the MNIST database.

When converting a QKeras model to an HLS project, the model quantization configuration is passed to hls4ml and enforced on the FPGA firmware. This procedure ensures that the use of specific arbitrary precision in the QKeras model is maintained during inference [1].

The resources at disposal on the FPGA are digital signal processors (DSPs), lookup tables (LUTs) and flip-flops (FF). The estimated resource consumption and latency from logic-synthesis, are listed in table 5.

Model	**Latency** [ns]	**Latency** [**clock period**]	**DSP** [%]	**LUT** [%]	**FF** [%]
AE	13	4.362	40 (4908)	10 (165311)	0.7 (22623)
AE pruned	13	4.362	33 (3989)	8 (139410)	0.6 (18900)
QAE pruned	10	4.362	0.4 (45)	6 (99424)	0.2 (5639)

Table 5: Model latency, resource utilization and relative energy estimate for three models on the Alveo U250. Resources are listed as percentage of total, with absolute numbers quoted in parenthesis. All the values reported here are computed using Vivado HLS tool after the hls4ml model conversion.

We compared the latency and resource consumption of the different models derived above for the classifications in the cases for: the *complete Autoencoder (AE)*, *compressed Autoencoder (AE pruned)* and *compressed Quantized Autoencoder (QAE pruned)*. We also compare the resource consumption simulation and latency for each model, targetting the *Alveo U250*.

Quantized Autoencoder with pruned classifiers is the most resource-efficient model, reducing the DSP usage by $\sim 99\%$, LUT usage by $\sim 40\%$ and the FF usage by $\sim 70\%$. The extreme reduction of DSP utilization is especially interesting as, on the FPGA, DSPs are scarce and usually become the critical resource for ML applications.

As we have reached a small resource usage during the synthesis phase we attempt to target a smaller FPGA, the Virtex-7 690T. You can see simulation results for the different FPGA target in the table 6 below.

Model	**Latency** [ns]	**Latency** [clock period]	**DSP** [%]	**LUT** [%]	**FF** [%]
QAE pruned	31	4.890	1.3 (45)	28 (121617)	3.8 (32562)

Table 6: Model latency, resource utilization and relative energy estimate for three models on the Virtex-7 690T. Resources are listed as percentage of total, with absolute numbers quoted in parenthesis. All the values reported here are computed using Vivado HLS tool after the hls4ml model conversion.

7 Conclusion

The aim of the paper is to analyze a new data processing strategy based on the implementation of an Autoencoder, an unsupervised Machine Learning technique, and its classification inference. As compared with traditional inference algorithms running on software, ML-based models implemented onto Field Programmable Gate Arrays promise shorter latency with a relatively small loss in accuracy.

To create models suited for hardware implementation, two optimizing techniques were performed: *weight pruning*, which reduced the number of multiplication needed when the NNs are used for inference, by setting, during the training procedure, the weights with the lowest values equal to 0, i.e. deleting the terms with negligible contributions.

The second way of optimizing the NNs was to make all the parameters inside the models *quantized*, i.e. represented as normalized fixed-point numbers with defined bitwidth. For this reason, we have used a novel library, QKeras, providing a simple method for uncovering optimally heterogeneously quantized Deep Neural Networks for a set of given resource or accuracy constraints. Quantization-aware models can be defined and trained by substituting Keras layers with models with heterogeneous per-parameter, and per-layer type precision. A model optimization algorithm which considers both model area and accuracy is presented, allowing users to maximize the model performance given a set of resource constraints, crucial for high-performance inference on edge. Taking a pre-trained model and making it suitable for hardware implementation, both in terms of latency and size, is one of the bottlenecks for bringing ML applications into extremely constrained computing environments, e.g. a detector at a particle collider. The workflow presented here will allow for a streamlined and simple process, ultimately improving the quality of physics data collected in future applications.

References

[1] Claudionor N Coelho et al. “Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors”. In: *Nature Machine Intelligence* 3.8 (2021), pp. 675–686.

[2] Javier Duarte et al. “Fast inference of deep neural networks in FPGAs for particle physics”. In: *Journal of Instrumentation* 13.07 (2018), P07027.

[3] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[5] Pankaj Mehta et al. “A high-bias, low-variance introduction to machine learning for physicists”. In: *Physics reports* 810 (2019), pp. 1–124.

[6] Lorenzo Valente. *AE4FPGA*. Version 1.0.4. May 2022. URL: `https://github.com/LorenzoValente3/Autoencoder-for-FPGA`.