Journal of Computer Science

Original Research Paper

Quantization and Pipelined Hardware Implementation of Deep Neural Network Models

El Hadrami Cheikh Tourad and Mohsine Eleuldj

Department of Computer Science, École Mohammedia d'Ingénieurs (EMI), Mohammed V University in Rabat, Morocco

Article history
Received: 24-06-2022
Revised: 14-08-2022
Accepted: 26-09-2022

Corresponding Author:
El Hadrami Cheikh Tourad
Department of Computer Science, École Mohammedia d'Ingénieurs (EMI), Mohammed V University in Rabat, Morocco
Email: elhadrami_cheikhtourad@research.emi.ac.ma

Abstract: In recent years, Deep Neural Networks (DNNs) have garnered much interest due to advances in computational power and data availability. Indeed, DNNs presents a considerable advantage in several challenges, such as classification problems and video analysis. Although, such accomplishment leads to significantly increasing energy demands, computational expenses, and memory capacity. In addition, current efficient DNNs may have more complex and extensive structures. As a result, implementing these huge models on embedded systems with limited sources is challenging. However, several works have attempted to solve the implementation issues while maintaining optimum accuracy. Among these ideas is compressing the model size using the quantization method and deploying it on Field Programmable Gate Arrays (FPGA) to enhance the latency and minimize the energy cost. This article presents a model optimizer using quantization methods to ensure the model hardware implementation. This optimizer compresses the model size and is integrated with a design flow that implements the model on the hardware. Furthermore, this article presents "DNN2FPGA," a design flow that can automatically implement the Deep Learning models on FPGA by producing pipelined HDL codes. This article indicates an excellent performance by decreasing the model's size and latency by 4x while maintaining the model's accuracy. It also presents a full review of the state of the art.

Keywords: DNN, Design Flow, Quantization, FPGA, Pipeline

Introduction

Deep Neural Networks (DNNs) have garnered much interest due to advances in computational power and data availability. Indeed, DNNs presents a considerable advantage in several challenges, such as classification problems and video analysis. Although, such accomplishment leads to significantly increasing energy demands, computational expenses, and memory capacity. In addition, current efficient DNNs may have more complex and extensive structures (nodes and layers). Figure 1 demonstrates that as the precision of the numbers utilized rises, so does the relative energy cost. For example, 8-bit addition costs 0.03 pJ, but 32-bit floatingpoint addition costs 0.9 pJ, or 30 times higher (Ducasse *et al*., 2021).

These days, DNNs may have more complex and extensive structures. For example, in DNNs, the number of layers may scale into tens of thousands and there can be billions of parameters (Ghimire *et al*., 2022. Consequently, deploying such huge models on real-time integrated circuits is defiance. In addition, these devices come with limited resources (energy, memory, and bandwidth), so there is an urgent need to find solutions for effectively deploying DNNs in low-powered devices (such as smartphones, embedded gadgets, and FPGA) without compromising model accuracy.

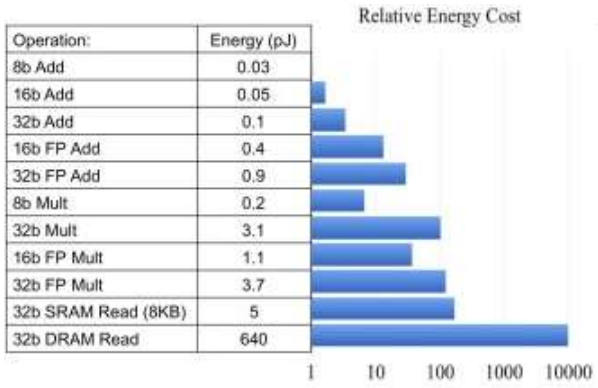

Fig. 1: Energy cost of operations in certain representations (Ducasse *et al*., 2021)

However, several works have attempted to minimize the memory and processing needs of DNNs while maintaining optimum accuracy to overcome limited device DNN implementation issues. Among these ideas is compressing the model size using the quantization method and deploying it on Field Programmable Gate Arrays (FPGA) to enhance the accuracy and minimize the energy cost.

Furthermore, FPGAs are suitable for DNN implementation (Tourad and Eleuldj, 2020). Because of their energy consumption efficiency, reconfigurability and computing capacity.

This article presents a model optimizer using quantization methods to ensure the model hardware implementation. This optimizer is integrated with the "DNN2FPGA" (Tourad and Eleuldj, 2021) design flow to compress the model size. Furthermore, this version of the design flow can automatically implement the Deep Learning models on FPGA by producing pipelined HDL codes.

This study structure contains six sections: The second section describes the background of DNN quantization. Next, the third section highlights the leading works in the DNN quantization method and reviews many related works to the DNNs hardware implementation. Then, the fourth section explains the proposed design flow and describes the quantization approach used in study work. After that, the fifth section shows the relevant results. Finally, the last section contains the conclusion.

This study includes the following contributions:

- The new model optimization approach includes three quantization methods
- New generic design flow integers a model optimizer and can automatically implement the Deep Learning models on FPGA by producing pipelined HDL codes
- Implement a CNN (Convolutional Neural network) for MNIST dataset classification to test and validate the design flow and the quantization method
- We discuss the obtained relevant results

DNN Quantization

Quantization is among the important and generally used model compression methods. For example, standard development frameworks frequently represent a neural network's parameters (bias, activations, and weights) as floating-point data. Recent studies have attempted to substitute this format with low-bit floating-point values or a small set of trained values. (Shawahna *et al*., 2018).

Quantization decreases computations by decreasing the accuracy of the data type, which reduces the bit width of the deep neural network's data storage and flow. Conversely, the computation and storing of data at a minor bit width permits fast inference while conserving energy. There are two types of quantization: Uniform and non-uniform quantization.

Uniform quantization presents the mapping function from real values to integer values, where the quantization levels are uniformly spaced (Gholami *et al*., 2021). Other research in state of the art investigates non-uniform quantization (Cai *et al*., 2017; Jeon *et al*., 2020; Faraone *et al*., 2018; Jung *et al*., 2019; Liao *et al*., 2020), where the widespread distribution is logarithmic and the quantization steps and levels are not equally distributed. However, this study focuses only on uniform quantization:

$$Q(r) = Int(r / S) - Z \tag{1}$$

Equation 1 (Gholami *et al*., 2021) defines the popular function to quantize the DNN, while Q is the quantization function, r denotes a real-valued parameter (bias, activation, or weight) and S denotes a real-valued scaling factor. Z denotes zero-point (quantization bias or offset) and Int is the float-to-integer rounding function. The scaling factor S converts the floating-point values to the corresponding low-precision values. For example, Zeropoint Z is a number with a low precision that reflects a quantized value that will represent the float value 0. The utility of zero-point is that we may have a broader range of integer values even for divergent tensors.

Calibration

Let [α, β] indicate the quantization range of real values and b presents the bit-width of the signed integer format (Gholami *et al*., 2021). Then, uniform quantization converts the input value x ϵ [α, β] to fit within the range $[-2^{b-1}; 2^{b-1} - 1]$, where all the values beyond this range are truncated to the closest limit. As a result, to specify the scaling factor, the clipping range [α, β] must first be specified. Calibration refers to the procedure of determining the clipping range. The most critical element in the quantization is the selection of the S. Equation 2 computes the scaling factor S:

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2}$$

A primary option is to determine the clipping range based on the minimum and maximum values of the signal; in this case, $\alpha = r_{min}$ and $\beta = r_{max}$. Given that the clipping range is not always symmetric concerning the origin, i.e., $-\alpha \neq \beta$, this method constitutes an asymmetric quantization technique. However, employing a symmetric quantization approach is also feasible by selecting a symmetric clipping range of $\alpha = -\beta$ and the zero-point z = 0, resulting in the same output as the previous example. Therefore, one common strategy for determining these parameters is to choose them based on the minimum and maximum values of the signal, as follows: $-\alpha = \beta = \max(|r_{max}|, |r_{min}|)$.

Furthermore, there are two activations quantization methods, dynamically and statically. Dynamic quantization calculates each activation map's range at runtime. This method involves real-time data statistics computation (max, min, and percentile.), which can be difficult. However, dynamic quantization is more accurate than static quantization since each input's signal range is computed. Static quantization pre-calculates and statically infers the clipping range. This method adds no computing complexity, although it is less accurate than dynamic quantization.

Yao *et al*. (2021) One of the most common approaches to pre-calculation is to compute the usual range of activations by running a series of calibration inputs.

Multiple metrics have been suggested to find the ideal range to reduce the Mean Squared Error (MSE) between the original parameters format and the quantized values corresponding to it. (Choukroun *et al*., 2019; Zhao *et al*., 2019). Although MSE is the most frequent way, one might consider utilizing other measures, such as entropy (Park *et al*., 2017). Another method is learning or applying this clipping range while the neural network is being trained. Works such as LQNets (Zhang *et al*., 2018) and LSQ+ (Bhalgat *et al*., 2020) are particularly noteworthy since they jointly optimize both the clipping range and the weights in DNN as it is being trained.

Fine-Tuning Methods

After the quantization process, it is frequently required to make adjustments to the parameters in the DNN. This update may be accomplished by retraining the model, a process referred to as Quantization-Aware Training (QAT), or it can be achieved without retraining, a process that is sometimes referred to as Post-Training Quantization (PTQ). As the name indicates, PTQ applies quantization to the model after it has been entirely trained using the Floating-Point 32 (FP32) weights and activations.

During training the model with QAT, the quantization loss is viewed as a part of the training loss. In most cases, QAT results in a more accurate model than PTQ, although PTQ is simple to implement.

Extreme Quantization

Quantization with very low bit accuracy is considered an extremely promising research subject. On the other hand, current approaches often result in a significant loss of accuracy compared to the baseline unless an extensive tuning and hyperparameter search is conducted. However, this accuracy loss can be tolerable for applications that are not as important. The most intensive form of quantization is binarization and it involves restricting the quantized values to a representation that uses just one bit.

As a result, the needed memory is cut down by a factor of 32. Furthermore, bitwise arithmetic may frequently execute binary (1-bit) and ternary (2-bit) computations, significantly improving over higher precisions such as FP32 and INT8. Besides the memory advantages that bitwise arithmetic offers. Some of the most often used binary neural networks are BinaryConnect (Courbariaux *et al*., 2014; 2015), Binarized Neural Network (BNN) (Courbariaux *et al*., 2016), and XNOR-Net (Rastegari *et al*., 2016; Bulat and Tzimiropoulos, 2019). We will go more into them in the next section.

Related Works

Various prior work – hardware accelerators – including hls4ml (Duarte *et al*., 2018), DL2HDL (Wielgosz *et al*., 2019), FP-DNN (Guan *et al*., 2017), and SysArrayAccel (Wei *et al*., 2017) support both fixedpoint and floating-point and also apply uniform quantization to all layers. Most operations in Finn (Ducasse *et al*., 2021.) are binary and focus on binary neural networks. Other frameworks, such as ALAMO (Ma *et al*., 2018), Auto Code Gen (Liu *et al*., 2016), and Angle-Eye (Wei *et al*., 2017), enable automated dynamic quantization for all layers at the time of compilation. Tourad and Eleuldj, 2020 studied several quantization-based frameworks for accelerating DNN models on FPGAs.

The results of (Wu *et al*., 2020) study demonstrate that the precision parameters of FP32 may be lowered to INT8 without causing a substantial reduction in the network accuracy. Another technique (Banner *et al*., 2019) created a 4-bit post-training quantization strategy that does not need the model to be fine-tuned after the quantization process. In (Jacob *et al*., 2018), only INT8 was employed for the training and inference of the ResNet-50 model, resulting in an accuracy loss of 1.5 percent. The research presented in reference (Hubara *et al*., 2017) generalizes the idea of bit precision, which allows storing weights and activations using any number of bits rather than only INT8.

Moreover, several studies focus on the binarization strategy. Significant work in this subject is Binary Connect (Courbariaux *et al*., 2015), which restricts weights to +1 or -1. This method maintains the weights as real values and is binarized solely during the forward and reverse runs to approximate the binarization effect. During the forward pass, the weights with real values are changed to +1 or -1 depending on the sign function. BNN (Courbariaux *et al*., 2016) expands on this notion by binarizing both the activations and the weights. The added advantage of binarizing weights and activations together is reduced latency since the expensive floating-point matrix multiplications may be substituted with lightweight XNOR operations followed by bit counting.

Another intriguing study (Rastegari *et al*., 2016; Bulat and Tzimiropoulos, 2019) is XNOR-Net, which achieves more precision by integrating a scaling factor into the weights. During training, XNOR-Net (Rastegari *et al*., 2016) and DoReFa (Zhou *et al*., 2016) sought to decrease quantization errors to expedite the training process. DoReFa-Net is a technique for training convolutional

neural networks with low-bit-width weights and activations using low-bitwidth parameter gradients (Zhou *et al*., 2016). Before being transmitted to convolutional layers, parameter gradients are stochastically quantized to low bit-width integers during the backward pass. Furthermore, forward/backward convolutions may now work with low bit-width weights. Training and inference may be accelerated using bit convolution kernels with DoReFaNet. AdaBits (Qin *et al*., 2020), another recent work based on the same DoReFa-Net concept using the adaptive quantization method. This approach proposes training a single model with multiple width multipliers for instant application adaptation (Qin *et al*., 2020).

AdaBits adjusts width, depth, and kernel sizes to improve predicted accuracy within the same computational restrictions.

Recently, IRNet, a binarization technique described by Qin *et al*. (2020) that uses a self-adaptive error-decay estimation to reduce gradient error in the learning phase, is the first method to handle data holding for both forward and backward propagation (Qin *et al*., 2020). To overcome these challenges, they propose an Information Retention Network (IR-Net) preserve information consisting of forwarding activations and backward gradients.

However, the frameworks in the research (Tourad and Eleuldj, 2020), among them the previously mentioned tools, are either non-generic (particular tool or equipment or limited templates) or comprise a substantial number of steps and tool flows, or use an extreme quantization which affects the model' accuracy.

Design Flow and Quantization Approach

(Tourad and Eleuldj, 2021) have recently suggested the "DNN2FPGA." A design flow employs a technique of direct hardware mapping.

This study presents a model optimizer using quantization methods to compress the model before the hardware implementation. Furthermore, this optimizer is integrated with a new version of the "DNN2FPGA" design flow with pipelined hardware implementation approach.

As illustrated in the design flow scheme (Fig. 2), the "DNN2FPGA" starts from the component Deep Learning Engine (DLE), in which the developer may create the DNN using a high-level framework such as Tensorflow or Keras. After that, the DLE sends the model to the optimizer component to apply a quantization method to reduce the parameter's bit-width. This method gives a gain in terms of memory and computation time. Then return the optimized model to the DLE. The DLE then generates a hierarchic representation of the model description using the Keras library.

The type of layer (pooling, convolutional, or fully connected), activation functions (ReLU, Tanh, and Sigmoid), and the layer calculations are then retrieved by the parser component. The model parser then retrieves the parameter (weights and biases).

Finally, these features are stored as configuration files and sent to the HDL generator.

The component HDL generator is responsible for many essential tasks: The timing stage is in charge of synchronizing the operation and establishing the clock cycle for the master-slave D-flip-flops. Moreover, this phase also handles the pipeline across layers. Finally, it separates calculations and memory access into clock cycles and distributes them to hardware units.

The layers are interconnected. Consequently, operating all network layers in the parallel mod is impossible.

Every layer should finish treating its current input (generate an output value) while accepting a new set of inputs to avoid overwriting internal registers during processes.

As illustrated in Fig. 3, these master-slave D-flipflops maintain the information till the clock pulse or signal, transferring it to the following layer and obtaining new data flow. This method increases the network throughput and enables the pipelined procedure.

The synthesis component then evaluates and compiles the generated HDL code. This stage first validates the HDL syntax and process statements. Next, the synthesis tool transforms the high-level representation into low-level representations by creating the bitstream file. Last, the synthesis confirms that the HDL code matches the hardware device.

While implementing the HDL on the device and ensuring that the results are equivalent to the software implementation, a simulation phase is necessary to ensure the HDL results are accurate. If the acquired results are below the software implementation, the complete process illustrated in Fig. 2 will be repeated, beginning with the generation of the model description. Otherwise, the procedure will proceed to load the bitstream files on the FPGA device. The validation parameters are the model precision and the latency (delay time); latency is the time necessary to estimate the test results for each deployment (HDL and software).

The bitstream file created by the synthesis component is then loaded on the designated device, especially in nonvolatile memory. This file contains the functionality, routing, and register default values. It also provides all the hardware components and instructions for controlling the device's assets. After downloading this file, the device is now prepared to use and test the Deep Learning model.

Comparing both hardware and software results is essential to validate the preciseness of the generated model. Therefore, after the model deployment on the device is completed, the model precision evaluation is built on the testing data (exploitation mode) to determine whether the generated results resemble the software results; if they do not, the entire process is restarted from the obtention of the model description file.

When it comes to deploying DNNs on FPGAs, two approaches may be used. First, starting from scratch, where the implementation handles training and inference,

or using a pre-trained model, where the hardware is just utilized for inference.

The DNN2FPGA can implement both, but we tested the design flow with a pre-trained model in our case.

Quantization Approach

The optimizer takes the model as input and then applies a symmetric quantization using a variety of calibrations. The optimizer strategy combines three different quantization methods applied one after the other until the necessary level of accuracy is achieved. The optimizer will begin by performing a post-training quantization on all layers. Next, it will test the model's accuracy and if it does not attain the desired accuracy, it will perform a uniform quantization on a portion of the neural network. This partial quantization leaves the more contributed layers, such as convolutional layers, in floating format. After that, do a new evaluation of the model's accuracy; if it does not yield the desired accuracy, go on to the process's Quantization-Aware Training (QAT) to fine-tune the model's accuracy. Finally, the training is deployed on the quantized model in the precedent phase with the same calibration method. This quantization approach uses a broad range of calibrations (min, max, percentile.).

Algorithm 1 explains all the required steps, from choosing the calibration and the accuracy verification. The algorithm is generic, which means it is independent of the bit-width (16-bit, 8bi. etc.) and the calibration method. First, the algorithm takes the DNN model as input and the user gives the quantization parameters (bitwidth and the calibration range). Then we browse all the layers and for each layer, we quantify every parameter using equation 1 in the second section. Next, we evaluate the model accuracy and, if not the desired one, continue to a partial quantization by skipping the sensitive layers, which in our case is the convolutional layer (the condition in the if clause). Finally, we apply a QAT method to finetune the model if the required accuracy does not yield.

Figure 4 presents the algorithm flowchart and explains the quantization steps.

Algorithm 1: Quantization Algorithm

```
Require: DNN model
Ensure: a quantized model
Chose the quantization bit-width
Chose the Calibration method
Calculate the scaling factor
# First, the post-training quantization for all layers
For L in Layers do
    For P in Parameter do
        # Apply the quantization equation to P
        P = Q(P)
        End For
    End For
  Evaluate the model accuracy
 IF the accuracy < desired accuracy then
 # Quantization for the non-sensitive layers
 For L in Layers do
    # Skip the sensitive layers
    IF L ! = convolutional layer
       For P in Parameter do
          # Apply the quantization equation to P
          P = Q(P)
       End For
    END IF
 End For
 Evaluate the model accuracy
 IF the new accuracy < desired accuracy then
    Apply the QAT method to fine-tune the model
 End IF
End IF
```

Case Study

This part represents the case study demonstrating how the design flow implements the quantization and tests the optimizer functionality. The design flow and the optimizer are implemented using Tensorflow/Keras framework. In this case study, we implement only the Int8 quantization to test the model optimizer inspired by the precedents works.

The CNN tested for the inference and prediction of a classification problem consists of an input layer, then two 2D convolutional layers with twelve filters using the ReLU (Rectified Linear Unit) as a non-linear function followed by a Max-pooling layer and a fully connected output layer with ten nodes.

Table 1 regroups All the network hyperparameters. Hyperparameters are parameters that are established before the model training process begins. Therefore, the values provided for these hyperparameters can affect the model learning rate and other parameters during training and final model performance.

The model error function is cross-entropy, commonly utilized to solve binary classification problems and the Adam method is an optimizer. Adam is an efficient form of the famous gradient descent algorithm used to solve various problems. The number of epochs in Table 1 presents the number of times the network is shown the entire training data while training. The batch size in Table 1 is the number of sub-samples sent to the network, after which the parameters are updated.

Table 1: Model hyperparameters

Parameters	Configuration
Loss function	Cross entropy
Optimizer	Adam
No of epoch	10
Batch size	500

Table 2: Quantization results

	Baseline model	Quantized model
Precision	FP32	Int 8
Memory	0.08 Mb	0.023 Mb
Accuracy	0.9655	0.9650
Latency	5.0030 s	1.230 s

DOI: 10.3844/jcssp.2022.1021.1029

This study uses the MNIST dataset of handwritten digits widely used to test image processing methods. It has a 60,000-example training set and a 10,000-example test set with 28 × 28-pixel samples.

Table 2 results demonstrate that the quantization reduces 4x the model size without losing the model's accuracy. Moreover, demonstrate that this quantization reduces the model latency by 4x. These results are obtained after implementing post-training and quantization-aware training to fine-tune the model's accuracy.

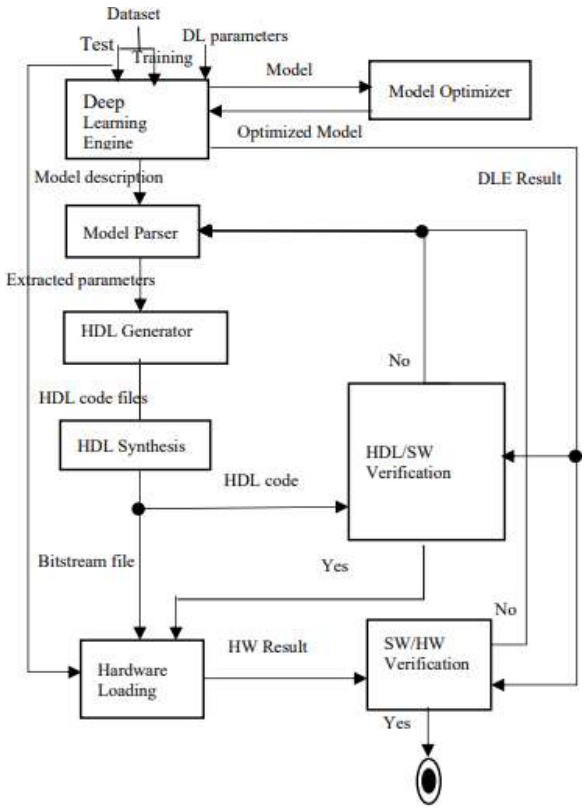

Fig. 2: The new design flow

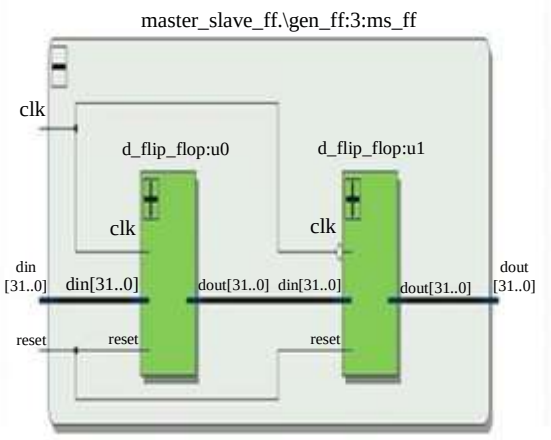

Fig. 3: Master-slave d-flip-flop

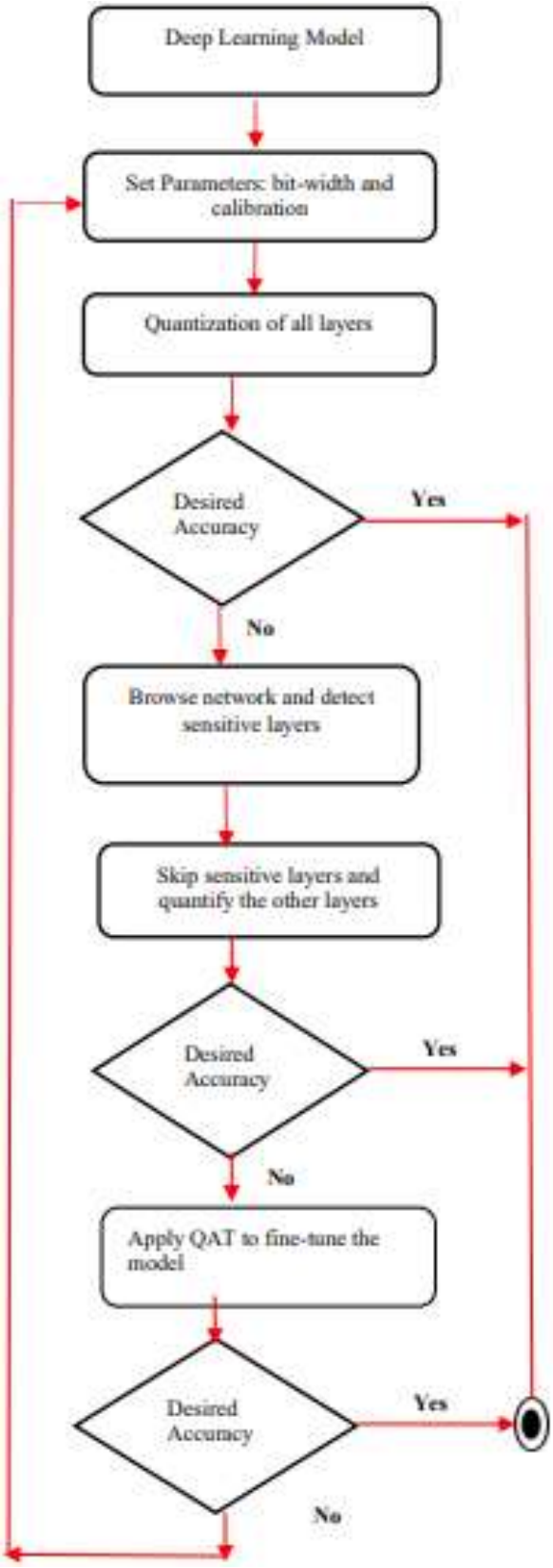

Fig. 4: Algorithm flowchart

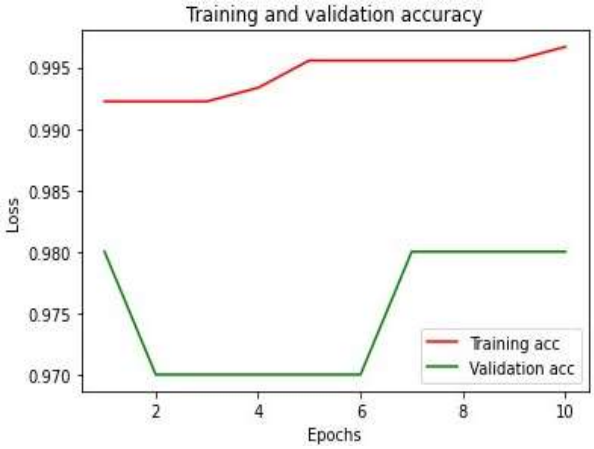

Fig. 5: QAT Training and validation accuracy

Figure 5 shows the accuracy and validation accuracy during all the training epochs.

Moreover, this figure shows the functionality of the model optimizer and tests the feasibility of this approach and the optimizer integration in the design flow. The results demonstrate promoted performance to improve in further works.

Conclusion and Future Works

This article presents a model optimizer using quantization methods to ensure the model hardware implementation. This optimizer is integrated with the "DNN2FPGA" design flow to compress the model size. Furthermore, this version of the design flow can automatically implement the Deep Learning models on FPGA by producing pipelined HDL codes.

This study content describes the background of DNN quantization. Next, it highlights the leading works in the DNN quantization method and reviews many related works to the DNNs hardware implementation. Then, it explains the proposed design flow and describes the quantization approach used in this study. After that, it shows the relevant results.

This article indicates an excellent performance by decreasing the model's size and latency by 4x while maintaining the model's accuracy. It also presents a full review of the state of the art.

This study concentrates on inference, meaning the implementation concentrates on exploiting the model. Further work is implementing the model from scratch (training and inference). Furthermore, we will improve the implementation, implement more optimization methods and deep learning models, evaluate metrics such as energy consumption and handle more significant network concerns.

Author's Contributions

El Hadrami Cheikh Tourad: Designed flow conception, data collection and implementation, analysis, interpretation of results, and manuscript preparation.

Mohsine Eleuldj: Supervised this study, and verified the analytical methods, analysis, and interpretation of results and manuscript preparation.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Banner, R., Nahshan, Y., & Soudry, D. (2019). Post-training 4-bit quantization of convolutional networks for rapid deployment. *Advances in Neural Information Processing Systems*, *32*. ttps://proceedings.neurips.cc/paper/2019/hash/c0a62e133894cdce435bcb4a5df1db2d-Abstract.html

Bhalgat, Y., Lee, J., Nagel, M., Blankevoort, T., & Kwak, N. (2020). Lsq+: Improving low-bit quantization through learnable offsets and better initialization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops* (pp. 696-697).

Bulat, A., & Tzimiropoulos, G. (2019). Xnor-net++: Improved binary neural networks. *arXiv preprint arXiv:1909.13863*. https://arxiv.org/abs/1909.13863

Choukroun, Y., Kravchik, E., Yang, F., & Kisilev, P. (2019, October). Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)* (pp. 3009-3018). IEEE. https://ieeexplore.ieee.org/abstract/document/9022167

Courbariaux, M., Bengio, Y., & David, J. P. (2014). Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*. https://arxiv.org/abs/1412.7024

Courbariaux, M., Bengio, Y., & David, J. P. (2015). Binary connect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, *28*. https://proceedings.neurips.cc/paper/2015/hash/3e15cc11f979ed25912dff5b0669f2cd-Abstract.html

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*. https://arxiv.org/abs/1602.02830

Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., ... & Wu, Z. (2018). Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, *13*(07), P07027. https://iopscience.iop.org/article/10.1088/1748-0221/13/07/P07027/meta

Ducasse, Q., Cotret, P., Lagadec, L., & Stewart, R. (2021). Benchmarking Quantized Neural Networks on FPGAs with FINN. *arXiv preprint arXiv:2102.01341*. https://arxiv.org/abs/2102.01341

Faraone, J., Fraser, N., Blott, M., & Leong, P. H. (2018). Syq: Learning symmetric quantization for efficient deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4300-4309).

Ghimire, D., Kil, D., & Kim, S. H. (2022). A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration. *Electronics*, *11*(6), 945. https://doi.org/10.3390/electronics11060945

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., & Keutzer, K. (2021). A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*. https://arxiv.org/abs/2103.13630

Guan, Y., Liang, H., Xu, N., Wang, W., Shi, S., Chen, X., ... & Cong, J. (2017, April). FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 152-159). IEEE. https://ieeexplore.ieee.org/abstract/document/7966671

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2017). Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, *18*(1), 6869-6898. https://www.jmlr.org/papers/volume18/16-456/16-456.pdf

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., ... & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2704-2713).

Jeon, Y., Park, B., Kwon, S. J., Kim, B., Yun, J., & Lee, D. (2020, November). Biqgemm: Matrix multiplication with lookup table for binary-coding-based quantized dnns. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-14). IEEE. https://ieeexplore.ieee.org/abstract/document/9355306

Jung, S., Son, C., Lee, S., Son, J., Han, J. J., Kwak, Y., ... & Choi, C. (2019). Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4350-4359).

Liao, Z., Couillet, R., & Mahoney, M. W. (2020). Sparse quantized spectral clustering. *arXiv preprint arXiv:2010.01376*. https://arxiv.org/abs/2010.01376

Liu, Z., Dou, Y., Jiang, J., & Xu, J. (2016, December). Automatic code generation of convolutional neural networks in FPGA implementation. In *2016 International conference on field-programmable technology (FPT)* (pp. 61-68). IEEE. https://ieeexplore.ieee.org/abstract/document/7929190

Ma, Y., Suda, N., Cao, Y., Vrudhula, S., & Seo, J. S. (2018). ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler. *Integration*, *62*, 14-23. https://doi.org/10.1016/j.vlsi.2017.12.009

Park, E., Ahn, J., & Yoo, S. (2017). Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5456-5464).

Qin, H., Gong, R., Liu, X., Shen, M., Wei, Z., Yu, F., & Song, J. (2020). Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 2250-2259).

Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016, October). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision* (pp. 525-542). Springer, Cham. https://link.springer.com/chapter/10.1007/978-3-319-46493-0_32

Shawahna, A., Sait, S. M., & El-Maleh, A. (2018). FPGA-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, *7*, 7823-7859. https://ieeexplore.ieee.org/abstract/document/8594633

Tourad, E. H. C., & Eleuldj, M. (2021, November). Generic Automated Implementation of Deep Neural Networks on Field Programmable Gate Arrays. In *The Proceedings of the International Conference on Smart City Applications* (pp. 989-1000). Springer, Cham. https://link.springer.com/chapter/10.1007/978-3-030-94191-8_80

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., ... & Cong, J. (2017, June). Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017* (pp. 1-6). https://doi.org/10.1145/3061639.3062207

Wielgosz, M., & Karwatowski, M. (2019). Mapping neural networks to FPGA-based IoT devices for ultra-low latency processing. *Sensors*, *19*(13), 2981. https://doi.org/10.3390/s19132981

Wu, H., Judd, P., Zhang, X., Isaev, M., & Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*. https://arxiv.org/abs/2004.09602

Yao, Z., Dong, Z., Zheng, Z., Gholami, A., Yu, J., Tan, E., ... & Keutzer, K. (2021, July). Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning* (pp. 11875-11886). PMLR. https://proceedings.mlr.press/v139/yao21a.html

Zhang, D., Yang, J., Ye, D., & Hua, G. (2018). Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 365-382).

Zhao, R., Hu, Y., Dotzel, J., De Sa, C., & Zhang, Z. (2019, May). Improving neural network quantization without retraining using outlier channel splitting. In *International conference on machine learning* (pp. 7543-7552). PMLR. http://proceedings.mlr.press/v97/zhao19c.html

Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., & Zou, Y. (2016). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*. https://arxiv.org/abs/1606.06160