Bridging QuantLab with LPDNN

Matteo Spallanzani, IIS, D-ITET, ETH Zürich
Cristian Cioflan, IIS, D-ITET, ETH Zürich
Miguel de Prado, Bonseyes Community Association

1 Introduction

Deep neural networks (DNNs) are the backbone of most contemporary artificial intelligence systems [1, 2, 3]. DNNs have considerable computational demands, requiring millions or even billions of parameters and a proportional amount of operations to process a single data point and deliver their accurate functionalities.

These properties make it challenging to deploy DNNs to resource-constrained devices such as embedded and edge computing platforms, and consequently to enable the pervasive application of artificial intelligence (AI). Several strategies have been proposed to reduce the computational requirements of DNNs, ranging from topological optimisations [4, 5, 6] to lower-level solutions such as tensor decomposition, parameter pruning, and operand quantisation [7, 8, 9].

Amongst these strategies, quantisation [9, 10] encompasses a series of techniques to replace the floating-point (FP) weights and features of DNNs with reduced-precision integerised counterparts. It is easy to see that quantising the operands of a DNN can significantly reduce its memory and storage footprints. For instance, moving from full-precision (i.e., 32-bit FP) down to single-byte integers can reduce these metrics by $4\times$. Reducing the memory and storage required by network operands also improves the arithmetic intensity of neural network inference, since it increases the amount of computational work that can be performed for a given amount of memory traffic. These quantised neural networks (QNNs) are particularly appealing when targetting embedded computing systems, since they are usually optimised around single instruction, multiple data (SIMD) integer arithmetic, and have limited or no support for FP arithmetic.

The machine learning engineering community has invested significant efforts to facilitate the deployment of DNNs to embedded and edge devices, as testified by a rich panorama of dedicated software tools [11, 12, 13, 14, 15, 16]. In this project, you will extend Low-Power Deep Neural Network (LPDNN) [11], a software framework capable of deploying DNNs to a diversified range of embedded, possibly heterogeneous computing platforms, to support new quantisation algorithms. You will achieve this functionality by connecting LPDNN to QuantLab [16], a software tool to train QNNs and prepare them for quantisation.

2 Background

2.1 Fake-quantised (FQ) arrays, true-quantised (TQ) arrays, and fake-to-true (F2T) conversion

To better understand the benefits of quantisation and how it is achieved in practice, consider the most important primitive of DNNs: the dot product. Let $\mathbf{x}$ and $\mathbf{w}$ be two arrays of FP numbers with $N \geq 1$ components each. Their dot product is

$$\langle \mathbf{x}, \mathbf{w} \rangle = \sum_{n=1}^{N} x_n w_n \,.$$

Evaluating this expression requires performing N FP products and N FP accumulations (if starting from an accumulator set to zero).

At training time, QNNs compute dot products between constrained arrays $\tilde{\mathbf{x}}, \tilde{\mathbf{w}}$ that can be written as

$$\tilde{\mathbf{x}} = \varepsilon_{\mathbf{x}} \hat{\mathbf{x}} \,, \tag{1}$$

$$\tilde{\mathbf{w}} = \varepsilon_{\mathbf{w}} \hat{\mathbf{w}} \,, \tag{2}$$

where $\varepsilon_{\mathbf{x}}, \varepsilon_{\mathbf{w}} > 0$ are FP *scale factors*, or simply *scales*, and $\hat{\mathbf{x}}, \hat{\mathbf{w}}$ are arrays whose components are integer and take value in some integer range

$$R(z, B) \coloneqq \{z + 0, z + 1, \ldots, z + 2^B - 1\} \,;$$

here, $B \geq 1$ is an integer *bit-width* and $z \in \mathbb{Z}$ is an integer *zero-point*. The arrays $\hat{\mathbf{x}}, \hat{\mathbf{w}}$ are called TQ arrays, since their components are truly integerised. The arrays $\tilde{\mathbf{x}}, \tilde{\mathbf{w}}$ are called FQ arrays, since their components are multiples of integers, but are represented using FP numbers. After training, one can map dot products between FQ arrays into dot products between TQ arrays by applying elementary arithmetic properties:

$$\begin{aligned}
\langle \tilde{\mathbf{x}}, \tilde{\mathbf{w}} \rangle &= \langle \varepsilon_{\mathbf{x}} \hat{\mathbf{x}}, \varepsilon_{\mathbf{w}} \hat{\mathbf{w}} \rangle \\
&= \sum_{n=1}^{N} \varepsilon_{\mathbf{w}} \hat{x}_n \varepsilon_{\mathbf{w}} \hat{w}_n \\
&= \sum_{n=1}^{N} \varepsilon_{\mathbf{x}} \varepsilon_{\mathbf{w}} \hat{x}_n \hat{w}_n \\
&= \varepsilon_{\mathbf{x}} \varepsilon_{\mathbf{w}} \left(\sum_{n=1}^{N} \hat{x}_n \hat{w}_n \right) \\
&= \varepsilon_{\mathbf{x}} \varepsilon_{\mathbf{w}} \langle \hat{\mathbf{x}}, \hat{\mathbf{w}} \rangle \,.
\end{aligned}$$

The last dot product only requires N integer multiplication and N integer accumulations (if starting from an accumulator set to zero), plus a single FP

multiplication. This rewriting is called F2T conversion [1].

Using the FQ format at training time is beneficial in that it allows to quickly train QNNs on mainframes and workstations where training kernels optimised for FP execution are available. It then suffices to perform F2T conversion before deploying the desired QNN to the target embedded platform [10].

2.2 Post-training quantisation (PTQ) and quantisation-aware training (QAT) algorithms

Creating QNNs is a non-trivial task. Indeed, discretising the parameters and features of a DNN layer most often creates discrepancies between the reference FP layer and its corresponding FQ counterpart. Given the compositional nature of DNNs, these discrepancies are propagated from a layer to the next, and the discrepancies intrinsic to the donwstream layers also add up. This propagation often results in errors in the outputs, destroying the functionality of the network that is being quantised. The algorithms that quantise neural networks must take into account these effects and can be partitioned into two families: PTQ algorithms, and QAT algorithms.

PTQ algorithms take as inputs FP DNNs that have been trained to convergence, and return FQ networks. As a first step, they introduce quantisers (i.e., operations that turn FP arrays into FQ arrays) into the target DNN; this process is known as float-to-fake (F2F) conversion. Then, the discrepancies between the FP and FQ network are analysed, possibly pushing some validation data points through both networks to get and compare statistics. Finally, the quantisers are adapted to minimise the discrepancies and ensure that the functionality of the target network is preserved. Note that no gradient descent step is performed [17].

On the other hand, QAT algorithms perform at least some gradient descent steps after F2F conversion [9, 18]. QAT algorithms can be applied either to pre-trained DNNs, in which case we might talk of quantisation-aware fine-tuning (QAFT), or to untrained networks.

PTQ algorithms usually work well for QNNs using 8-bit operands. QAT algorithms starting from pre-trained DNNs usually work well for QNNs using sub-byte operands and solving simple tasks. Targetting aggressively quantised operands (e.g., binary or ternary) usually requires training QNNs from scratch using QAT algorithms.

2.3 Open Neural Network eXchange (ONNX)

ONNX is a standard to describe DNNs [19]. Each version of ONNX defines and supports a collection of operations (e.g., convolutions, additions, activation functions) that are typically composed to create DNNs [20].

[1] Real F2T conversions are a bit more involved when batch-normalisation and activation functions are considered, but their goal (mapping operations between FQ arrays to operations between TQ arrays) and the tools used to achieve the goal (elementary arithmetic properties) remain the same

Popular deep learning frameworks (e.g., TensorFlow, PyTorch) include export utilities to map their idiosyncratic network representations to the corresponding ONNX ones. For instance, it is possible to train two DNNs using different deep learning frameworks, and still express them in a standardised format, making code generation and compilation framework-agnostic. By providing a unified description format for DNNs, ONNX decouples the creation of a network from its deployment.

2.4 QuantLab & QuantLib

QuantLab is a PyTorch-based software tool to train QNNs using QAT algorithms, tune the algorithms' hyper-parameters and prepare the networks for deployment [16]. QuantLab consists of two parts.

QuantLib QuantLib is a quantisation library [21]. It implements the building blocks to perform F2F conversion, train QNNs using several QAT algorithms, and perform F2T conversion.

QuantLab QuantLab is an experiment manager [16]. It implements a training environment to facilitate the structured exploration of QNN performance when applied to a variety of data sets and network architectures.

2.5 LPDNN

LPDNN is a framework to generate portable and efficient DNN-based machine learning systems. The goal of LPDNN is to provide a zoo of DNNs to solve a diversified range of tasks (e.g., object detection, image classification, speech recognition) and that can be optimised for and deployed embedded platforms featuring heterogeneous processing elements (central processing units (CPUs), general-purpose graphics processing units (GPGPUs), field-programmable gate arrays (FPGAs), digital signal processors (DSPs), application-specific integrated circuits (ASICs)). LPDNN models each solution as a so-called AI application, and achieves efficiency by delegating program optimisations and code generation to its modular LPDNN iNference Engine (LNE).

AI applications AI applications are the abstraction used by LPDNN to represent deployable deep learning solutions. A basic AI application consists of two modules: a pre-processing module preparing the raw inputs for the core processing (e.g., by normalising them) and a DNN module performing the core neural network processing. A basic AI application can be extended by including a post-processing module, which allows the creation of data structures such as bounding boxes or facial landmarks.

LNE At the heart of LPDNN lies LNE. LNE is a code generator aiming at accelerating the execution of neural networks on heterogeneous embedded platforms. LNE combines a plugin-based architecture with a dependency-free inference core that supports inference of DNNs on a wide range of embedded devices. LNE's core includes a set of CPU C++ functions that can be complemented by platform-specific acceleration libraries (e.g., ARM Compute Library (ARM-CL), CUDA) to generate code for AI applications that is optimised for the target platform.

LNE supports a wide range of neural network models as it provides direct compatibility with ONNX. First, the ONNX representation of a given network is converted to an LPDNN-specific computational graph format. Then, several steps such as PTQ and optimisations based on graph analysis (e.g., operator fusion, optimised memory allocation) are performed. Finally, each node of the computation graph is linked to the corresponding backend routine that will be called during the execution of the neural network. Runtime functions orchestrate the information flow between the backends executing the compiled network.

3 Project plan

The goal of this project is to extend LPDNN's network quantisation capabilities by providing it with a front-end capable of QAT training; in our case, QuantLab. The project will consist of two parts: in the first part, you will train a facial landmark localisation DNN in QuantLab using a QAT algorithm; in the second part, you will develop the required LPDNN software connectors to parse the trained model and deploy it to one of LPDNN's supported backends.

3.1 Train a facial landmark localisation network using a QAT algorithm

As a benchmark AI application, we will target facial landmark localisation [22, 23]. In particular, we will quantise the 3-dimensional dense face alignment (3DDFA) network architecture [24, 25] to eight bits using the parametrised clipping activation (PACT) QAT algorithm [18].

During this part of the project, you will first familiarise yourself with the problem of facial landmark localisation and with the 3DDFA network architecture; you will also familiarise yourself with QuantLab and QuantLib. Then, you will train 3DDFA using PACT, identifying suitable configurations of the training hyper-parameters that can preserve the network's functionality. Finally, you will export an ONNX representation of the trained network; remember: QuantLab-exported ONNX files include annotations that are not supported by the ONNX standard.

3.2 Develop software connectors to bridge QuantLab with LPDNN

As a benchmark platform, we will target an off-the-shelf Raspberry Pi 4 (RPI4) which includes a Cortex-A72 quad-core CPU.

During this part of the project, you will first familiarise yourself with LPDNN; in particular, you will familiarise yourself with its ONNX parser, its modular system of backends, and more specifically with its ARM-CL backend. Then, you will extend LPDNN's ONNX parser to accept QuantLab-exported ONNX files. Finally, you will extend LPDNN's ARM-CL backend to accept the newly-accepted ONNX parsings.

3.3 Bonus task: compare PTQ and QAT algorithms

If time remains, we will compare the functional performance of LPDNN networks created using its integrated PTQ algorithm and the newly-supported QAT algorithm.

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, T. Guez, A. Hubert, L. Baker, A. Lai, M. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, 2019.

[3] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, S. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Higly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, pp. 583–589, 2021.

[4] F. Chollet, "Xception: deep learning with depthwise separable convolutions," in *Proceedings of the 2017 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2017)*, IEEE, 2017.

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: inverted residuals and linear bottlenecks," in *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2018)*, IEEE, 2018.

[6] M. Tan and Q. V. Le, "EfficientNet: rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, MLResearchPress, 2019.

[7] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, pp. 455–500, 2009.

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proceedings of the 2016 International Conference on Learning Representations (ICLR 2016)*, International Conference on Learning Representations (ICLR), 2016.

[9] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2018.

[10] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2018)*, IEEE, 2018.

[11] M. de Prado, M. Denna, L. Benini, and N. Pazos, "QUENN: quantization engine for low-power neural networks," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ACM, 2018.

[12] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "TensorFlow Lite Micro: embedded machine learning on TinyML systems," 2020.

[13] `https://github.com/quic/aimet`, December 2020.

[14] "Vitis AI User Guide (v1.4)," July 2021.

[15] `https://www.arm.com/technologies/compute-library`, November 2021.

[16] `https://github.com/pulp-platform/quantlab`.

[17] R. Banner, Y. Nahshan, and D. Soudry, "Post-training 4-bit quantization of convolutional networks for rapid deployment," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS 2019)*, Neural Information Processing Systems, 2019.

[18] J. Choi, S. Venkataramani, V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," in *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML 2019)*, Conference on Systems and Machine Learning, 2019.

[19] `https://onnx.ai/`.

[20] `https://github.com/onnx/onnx/blob/main/docs/Operators.md`.

[21] `https://github.com/pulp-platform/quantlib`.

[22] M. Köstinger, P. Wohlhart, P. M. Roth, and H. Bischof, "Annotated facial landmarks in the wild: a large-scale, real-world database for facial landmark localization," in *Proceedings of the 2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, IEEE, 2011.

[23] C. Sagonas, E. Antonakos, G. Tzimiropoulos, S. Zafeiriou, and M. Pantic, "300 Faces In-The-Wild challenge: database and results," *Image and Vision Computing*, vol. 47, pp. 3–18, 2016.

[24] X. Zhu, Z. Lei, X. Liu, H. Shi, and S. Z. Li, "Face alignment across large poses: a 3D solution," in *Proceedings of the 2016 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2016)*, IEEE, 2016.

[25] J. Guo, X. Zhu, Y. Yang, F. Yang, Z. Lei, and S. Z. Li, "Towards fast, accurate and stable 3D dense face alignment," in *Proceedings of the 16th European Conference on Computer Vision (ECCV 2020)*, Springer, 2020.