Probing the limits of fake-quantised neural networks

Matteo Spallanzani, IIS, D-ITET, ETH Zürich
Renzo Andri, Huawei Technologies Switzerland AG

1 Introduction

The last decade has witnessed an unprecedented surge in the interest for machine learning systems, and in particular for deep neural networks (DNNs). DNNs have risen to prominence mostly due to two factors: the increase in quantity, size, and quality of labelled data sets, containing sufficient information to characterise complex relationships; and the increase in computing power due to programmable, parallel hardware, optimised to execute programs which are trivially parallelisable and have data-independent control flows.

Given a DNN "A" that solves a task, a machine learning engineer can improve its accuracy by changing the network topology to derive a more effective DNN "B". However, these changes often involve increasing the computational complexity, i.e., the number of network parameters and operations. When "B" has more parameters than "A", this implies that the computing system "T" executing "B" requires larger storage and memory than the computing system "S" executing "A". When "B" requires more operations than "A", if we suppose that both networks must process a given amount of data points in a reference time interval (i.e., that they must reach a target *application throughput*), "T" must be able to execute more operations per unit of time than "S", i.e., it must have higher *throughput* (Op/s). This reasoning shows that more powerful computational resources can be sufficient to scale up task accuracy; but are they necessary? And, more importantly, can we always assume that it is possible to scale up computational resources?

In many applications, constraints such as application latency and data privacy make it desirable to perform inference directly were the data is produced, often on embedded, mobile, and edge devices (i.e., Internet-of-things (IoT) nodes). These computing systems have limited storage and memory with respect to high-performance computing (HPC) clusters and workstations, where DNNs are typically executed. Moreover, they are often battery-powered, meaning that they can not exceed hard constraints on total energy consumption (J) or peak power (W). For this reason, the research field of TinyML has focussed on creating machine learning systems that are not only effective, but also efficient in terms of storage, memory and energy requirements.

Several strategies have been explored to improve the efficiency of DNNs and enable their deployment on resource-constrained computing devices. Topological optimisations have focussed on developing network topologies that are efficient in terms of accuracy-per-parameter or accuracy-per-operation [1]. Other strategies have focussed on reducing model size by means of techniques such as parameter pruning [2]. Finally, quantised neural networks (QNNs) have been proposed to replace high-bit-width, energy-costly floating-point operands with low-bit-width, energy-efficient integer operands, with the double benefit of reducing model size and replacing floating-point operations with much simpler integer operations [3]. Recent research has shown that DNN operands can be quantised to low-bit-width integers without deteriorating the system's performance [4]. These properties make QNNs an extremely good fit for embedded devices, which usually have limited storage and memory, have limited support for floating-point arithmetic, and are optimised for executing integer arithmetic using single instruction, multiple data (SIMD) ISA extensions.

QNNs are usually trained on HPC clusters and workstations to leverage the optimised software kernels provided by the most popular deep learning frameworks such as TensorFlow or PyTorch. At training time, QNNs use floating-point operands to exploit the efficiency of floating-point calculations on massively parallel accelerators, such as NVidia's GPGPUs; these operands are constrained in such a way to enable the conversion of these fake-quantised (FQ) networks into fully integerised programs that can be executed efficiently on embedded hardware at inference time. We name these integerised programs true-quantised (TQ) networks.

Unfortunately, converting a FQ network to a TQ one is a lossy process. Limiting such losses is a delicate but fundamental problem in QNN practice. In this project, we will explore the discrepancies between the FQ and TQ versions of QNNs that arise when using different floating-point representations during the FQ training stage.

Why do we use fake-quantisation?

In principle, we could use integer operands directly at training time. So why do we use FQ arrays when training QNNs? Using integer operands would require us to replace the optimised floating-point software kernels used by the chosen deep learning framework with counterparts that can handle integer data types. The cost of the design, implementation and testing activities implied by this choice would anyway not compensate the payoff: GPGPUs are optimised to carry out multiplications between large matrices of floating-point numbers, whereas it is significantly less efficient to perform integer operations. Moreover, it is usually assumed that training a QNN using integer arithmetic would generate numerically unstable gradients; however, to the best of our knowledge neither a formal nor an experimental analysis exists proving this point.

2 Background and problem definition

2.1 Feedforward DNNs

For simplicity, in this work we will only consider feedforward network topologies, i.e., DNNs for which the graph determined by its artificial neurons is acyclic. Examples of feedforward topologies are AlexNet, VGG, and MobileNetV1 [5, 6, 1].

From a functional perspective, feedforward DNNs are compositions of layer maps. Given $L \in \mathbb{N}, L > 1$, and a collection of positive integers $n^0, \dots, n^L$, each layer map takes the following form:

$$\begin{aligned} \varphi^\ell \,:\, X^{\ell-1} &\to X^\ell \\ \mathbf{x}^{\ell-1} &\mapsto \boldsymbol{\zeta}^\ell(\mathbf{W}^\ell \mathbf{x}^{\ell-1} + \mathbf{b}^\ell)\,, \end{aligned} \tag{1}$$

where $X^{\ell-1} \subseteq \mathbb{R}^{n^{\ell-1}}, X^\ell \subseteq \mathbb{R}^{n^\ell}$ are the *representation spaces* or *feature spaces*, $\mathbf{W}^\ell \in \mathcal{M}^{n^\ell \times n^{\ell-1}}(\mathbb{R})$ is the ℓ-th *weight matrix*, $\mathbf{b}^\ell \in \mathbb{R}^{n^\ell}$ is the *bias vector*, and $\boldsymbol{\zeta}^\ell : \mathbb{R}^{n^\ell} \to \mathbb{R}^{n^\ell}$ is a non-constant, non-linear *activation function.* In particular, $\boldsymbol{\zeta}^\ell$ is the element-wise application of n^ℓ non-constant functions $\zeta^\ell_{i^\ell} : \mathbb{R} \to \mathbb{R}, i^\ell = 1, \dots, n^\ell$, of which at least one is non-linear:

$$\boldsymbol{\zeta}^\ell(\mathbf{W}^\ell \mathbf{x}^{\ell-1} + \mathbf{b}^\ell) \coloneqq \left(\zeta^\ell_1(\langle \mathbf{w}^\ell_1, \mathbf{x}^{\ell-1}\rangle + b^\ell_1), \dots, \zeta^\ell_{n^\ell}(\langle \mathbf{w}^\ell_{n^\ell}, \mathbf{x}^{\ell-1}\rangle + b^\ell_{n^\ell})\right) \,.$$

In this formalism, a feedforward DNN is a composition of layer maps (1):

$$\Phi \coloneqq \varphi^L \circ \dots \circ \varphi^1 \,. \tag{2}$$

Modern feedforward networks, and especially convolutional neural networks (CNNs), apply the so-called *batch normalisation* transform in between the linear and non-linear parts of (1) to increase task accuracy [7]. Given vector parameters $\boldsymbol{\mu}^\ell, \boldsymbol{\sigma}^\ell, \boldsymbol{\gamma}^\ell, \boldsymbol{\beta}^\ell \in \mathbb{R}^{n^\ell}$, a batch-normalised layer map acts as follows:

$$\varphi^\ell(\mathbf{x}^{\ell-1}) = \boldsymbol{\zeta}^\ell \left(\left(\left(\mathbf{W}^\ell \mathbf{x}^{\ell-1} + \mathbf{b}^\ell - \boldsymbol{\mu}^\ell \right) /_\odot \boldsymbol{\sigma}^\ell \right) \times_\odot \boldsymbol{\gamma}^\ell + \boldsymbol{\beta}^\ell \right) \,;$$

here, $/_\odot$ represents component-wise division, whereas $\times_\odot$ represents component-wise multiplication. In particular, the bias vector is usually folded in the mean vector, yielding

$$\varphi^\ell(\mathbf{x}^{\ell-1}) = \boldsymbol{\zeta}^\ell \left(\left(\mathbf{W}^\ell \mathbf{x}^{\ell-1} \right) /_\odot \times_\odot \boldsymbol{\gamma}'^\ell + \boldsymbol{\beta}'^\ell \right) \,, \tag{3}$$

where $\boldsymbol{\gamma}'^\ell \coloneqq \boldsymbol{\gamma}^\ell /_\odot \boldsymbol{\sigma}^\ell$, and $\boldsymbol{\beta}'^\ell \coloneqq (-\boldsymbol{\mu}^\ell \times_\odot \boldsymbol{\gamma}^\ell + \boldsymbol{\beta}^\ell \times_\odot \boldsymbol{\sigma}^\ell) /_\odot \boldsymbol{\sigma}^\ell$.

If we consider a single artificial neuron, the neuron map acts as follows:

$$\varphi^\ell_{i^\ell}(\mathbf{x}^{\ell-1}) = \boldsymbol{\zeta}^\ell(\langle \mathbf{w}^\ell_{i^\ell}, \mathbf{x}^{\ell-1} \rangle \gamma'^\ell_{i^\ell} + \beta'^\ell_{i^\ell})\,, \tag{4}$$

where $i^\ell \in \{1, \dots, n^\ell\}$. To avoid overloading the notation unnecessarily, we will drop the neuron index i^ℓ when the distinction will be irrelevant.

2.2 Quantisers

Given $K \in \mathbb{N}, K > 1$, a *K-quantiser* (or simply *quantiser*) is a function

$$\varsigma(x) = \sum_{k=0}^{K-1} \chi_{I_k}(x) q_k \,,$$

where $Q \coloneqq \{q_0 < \cdots < q_{K-1}\} \subset \mathbb{R}$ is the collection of *quantisation levels*, χ_I is the indicator function of $I \subseteq \mathbb{R}$, and $\{I_0, \dots, I_{K-1}\}$ is a partition of $\mathbb{R}$ composed by non-intersecting, successive intervals (*bins*). In particular, we usually consider a set of *thresholds* $\Theta \coloneqq \{\theta_1 < \cdots < \theta_{K-1}\} \subset \mathbb{R}$ and define $I_0 \coloneqq (-\infty, \theta_1)$, $I_k \coloneqq [\theta_k, \theta_{k+1})$ for $k = 1, \dots, K-2$, and $I_{K-1} \coloneqq [\theta_{K-1}, +\infty)$. This definition shows that quantisers are piece-wise constant, monotonically increasing functions, and that they can be computed in two steps: *binning*, i.e., computing the bin index $k(x)$, and *dequantisation*, i.e., mapping $k(x)$ to the output value q_k. When $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$ is a multi-dimensional vector, we define the convention $\varsigma(\mathbf{x}) \coloneqq (\varsigma(x_1), \dots, \varsigma(x_n))$.

When the number K of bins is low, threshold-based operations can be implemented by means of look-up tables. However, when K is large the application of a threshold-based quantiser might become too costly. A simplification commonly used in literature is assuming that there exist $z_\Theta \in \mathbb{Z}, \epsilon_\Theta \in \mathbb{R}^+$ such that $\theta_k = \epsilon_\Theta(k + z_\Theta), k = 1, \dots, K-1$, then computing

$$k(x) = clip(\lfloor x/\epsilon_\Theta \rfloor - z_\Theta, 0, K-1) \,, \tag{5}$$

where $\lfloor t \rfloor \coloneqq \max\{i \in \mathbb{N} \,|\, i \leq t\}$ is the *flooring* operation and $clip(t, a, b) \coloneqq \max\{a, \min\{t, b\}\}, a, b \in \mathbb{R}, a < b$ is the *clipping* operation.

Given a *precision* $B \in \mathbb{N}$, we can link quantisers to integer arithmetic by considering $K = 2^B$ and assuming that exist $z \in \mathbb{Z}, \epsilon \in \mathbb{R}^+$ such that the dequantisation can be computed as

$$q_k = \epsilon(k + z), k = 0, \dots, K-1 \,. \tag{6}$$

z is called the *offset* or *zero-point* of the quantiser, whereas ϵ is its *scale* or *quantum*. The most common quantisers used in the literature fuse (5) and (6) by setting $z_\Theta = z$ and $\epsilon_\Theta = \epsilon$:

$$\varsigma_{z,\epsilon}(x) = \epsilon(clip(\lfloor x/\epsilon \rfloor, 0 + z, K - 1 + z)) \,. \tag{7}$$

We say that an array $\tilde{\mathbf{x}} \in \mathbb{R}^n$ is *fake-quantised* if there exist $z \in \mathbb{Z}, \epsilon_\mathbf{x} \in \mathbb{R}^+$ such that $\tilde{x}_i = \epsilon_\mathbf{x} \hat{x}_i, i = 1, \dots, n$, where $\hat{x}_i \in \{0 + z, \dots, 2^B - 1 + z\}$. The importance of fake-quantised arrays is quickly explained. Given two fake-quantised arrays $\tilde{\mathbf{w}}, \tilde{\mathbf{x}} \in \mathbb{R}^n$, we can use basic arithmetic properties to rewrite

$$\begin{aligned} \langle \tilde{\mathbf{w}}, \tilde{\mathbf{x}} \rangle &= \sum_{i=1}^{n} \epsilon_\mathbf{w} \hat{w}_i \epsilon_\mathbf{x} \hat{x}_i \\ &= \epsilon_\mathbf{w} \epsilon_\mathbf{x} \sum_{i=1}^{n} \hat{w}_i \hat{x}_i \\ &= \epsilon_\mathbf{w} \epsilon_\mathbf{x} \langle \hat{\mathbf{w}}, \hat{\mathbf{x}} \rangle \,. \end{aligned} \tag{8}$$

From the perspective of hardware arithmetic, a dot product between fake-quantised arrays can be transformed into a product between a floating-point scalar and the result of the dot product between integer arrays.

Given an array $\mathbf{x} \in \mathbb{R}^n$, we can get a fake-quantised array by applying a quantiser (7) to each component:

$$\tilde{\mathbf{x}} = \varsigma_{z,\epsilon}(\mathbf{x})$$

2.3 Fake-quantised and true-quantised neurons

Fake-quantised QNNs use specific neuron maps (4). In particular, they take as input a fake-quantised feature array $\tilde{\mathbf{x}}^{\ell-1} = \epsilon_{\mathbf{x}^{\ell-1}} \hat{\mathbf{x}}^{\ell-1}$, use a fake-quantised weight array $\tilde{\mathbf{w}}^{\ell} = \epsilon_{\mathbf{w}^{\ell}} \hat{\mathbf{w}}^{\ell}$, and apply a quantiser (7) with quantum $\epsilon_{\mathbf{x}^{\ell}}$ as their activation function, producing a fake-quantised number as output:

$$\tilde{x}^{\ell-1} = \epsilon_{\mathbf{x}^{\ell}} \operatorname{clip}\left(\left\lfloor \frac{\langle \tilde{\mathbf{w}}^{\ell}, \tilde{\mathbf{x}}^{\ell-1} \rangle \gamma'^{\ell} + \beta'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} \right\rfloor, a, b \right) .$$

FQ neuron maps are named "fake" because we can turn them into fully-integerised functions by means of basic arithmetic properties, in a process named fake-to-true (F2T) conversion. But how does this process work?

Consider the first argument of the clipping function. Thanks to the distributive property of real arithmetic, we can rewrite it as

$$\left\lfloor \frac{\langle \tilde{\mathbf{w}}^{\ell}, \tilde{\mathbf{x}}^{\ell-1} \rangle \gamma'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} + \frac{\beta'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} \right\rfloor .$$

Thanks to the definition of fake-quantised array, we can rewrite this as

$$\left\lfloor \frac{\langle \tilde{\mathbf{w}}^{\ell}, \tilde{\mathbf{x}}^{\ell-1} \rangle \gamma'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} + \frac{\beta'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} \right\rfloor = \left\lfloor \frac{\langle \epsilon_{\mathbf{w}^{\ell}} \hat{\mathbf{w}}^{\ell}, \epsilon_{\mathbf{x}^{\ell-1}} \hat{\mathbf{x}}^{\ell-1} \rangle \gamma'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} + \frac{\beta'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} \right\rfloor . \tag{9}$$

By the linearity of the dot product, we can rewrite the right-hand side as

$$\left\lfloor \frac{\langle \hat{\mathbf{w}}^{\ell}, \hat{\mathbf{x}}^{\ell-1} \rangle \epsilon_{\mathbf{w}^{\ell}} \epsilon_{\mathbf{x}^{\ell-1}} \gamma'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} + \frac{\beta'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} \right\rfloor ,$$

where we see that the dot product now involves only integer arrays. Thanks to the associative property of real arithmetic, we can further rewrite this as

$$\left\lfloor \langle \hat{\mathbf{w}}^{\ell}, \hat{\mathbf{x}}^{\ell-1} \rangle \frac{\epsilon_{\mathbf{w}^{\ell}} \epsilon_{\mathbf{x}^{\ell-1}} \gamma'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} + \frac{\beta'^{\ell}}{\epsilon^{\ell}_{\mathbf{x}}} \right\rfloor .$$

Now let $D \in \mathbb{N}$; given any positive real number $x \in \mathbb{R}^+$, we can observe that

$$\lim_{D \to +\infty} \frac{x2^D - \lfloor x2^D \rfloor}{2^D} = 0$$

because $x2^D - \lfloor x2^D \rfloor \in [0, 1)$ and $\lim_{D\to+\infty} 2^D = +\infty$. Therefore, the following holds true[1]:

$$\lim_{D\to+\infty} \left\lfloor \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1} \rangle \hat{\gamma}^\ell + \hat{\beta}^\ell \right) / 2^D \right\rfloor = \left\lfloor \langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1} \rangle \frac{\epsilon_{\mathbf{w}^\ell} \epsilon_{\mathbf{x}^{\ell-1}} \gamma'^\ell}{\epsilon^\ell_{\mathbf{x}}} + \frac{\beta'^\ell}{\epsilon^\ell_{\mathbf{x}}} \right\rfloor ,$$

where $\hat{\gamma}^\ell := \lfloor 2^D \epsilon_{\mathbf{w}^\ell} \epsilon_{\mathbf{x}^{\ell-1}} \gamma'^\ell / \epsilon^\ell_{\mathbf{x}} \rfloor$ and $\hat{\beta}^\ell := \lfloor 2^D \beta'^\ell / \epsilon^\ell_{\mathbf{x}} \rfloor$. If we choose D sufficiently large, we can therefore approximate

$$\left\lfloor \langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1} \rangle \frac{\epsilon_{\mathbf{w}^\ell} \epsilon_{\mathbf{x}^{\ell-1}} \gamma'^\ell}{\epsilon^\ell_{\mathbf{x}}} + \frac{\beta'^\ell}{\epsilon^\ell_{\mathbf{x}}} \right\rfloor \approx \left\lfloor \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1} \rangle \hat{\gamma}^\ell + \hat{\beta}^\ell \right) / 2^D \right\rfloor . \tag{10}$$

Note that $\hat{\gamma}^\ell, \hat{\beta}^\ell$ are integers. Also, supposing that t is a number represented in binary format, the operation $\lfloor t/2^D \rfloor$ can be implemented in digital hardware as a right-shift of t's representation by D positions, followed by a truncation of its decimal part. Since in (10) t is an integer, this operation corresponds to truncating its D least significant bits (LSBs).

We name the right-hand side in (10) the TQ representation of the neuron map (9). With some adaptations, the F2T conversion process can be scaled up to a full layer map (3), and from there to a complete network (2).

2.4 The problem of F2T conversion

The standard representation for real numbers in digital hardware is the floating-point numeric data format. Unfortunately, most real (and therefore also fake-quantised) numbers do not have an exact representation in the floating-point format. Moreover, floating-point arithmetic does not satisfy the most convenient properties that are enjoyed by real arithmetic: for instance, floating-point arithmetic is non-associative and non-distributive.

Most of the rewritings operated in the previous sub-section relied on the properties of real arithmetic. In practical F2T conversions, errors can be introduced at each step due to the violations of these properties on part of floating-point arithmetic. In its simplest form, our question is the following: how large can be the difference

$$\left\lfloor \frac{\langle \epsilon_{\mathbf{w}^\ell} \hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}} \hat{\mathbf{x}}^{\ell-1} \rangle \gamma'^\ell + \beta'^\ell}{\epsilon^\ell_{\mathbf{x}}} \right\rfloor - \left\lfloor \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1} \rangle \hat{\gamma}^\ell + \hat{\beta}^\ell \right) / 2^D \right\rfloor ? \tag{11}$$

Given $a, b \in \mathbb{R}^+_0, a < b$, the inequality $\lfloor b - a \rfloor \leq \lfloor b \rfloor - \lfloor a \rfloor$ always holds. Therefore, we can get a lower-bound for (11) by characterising the difference between the arguments of the floating point operations:

$$\frac{\langle \epsilon_{\mathbf{w}^\ell} \hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}} \hat{\mathbf{x}}^{\ell-1} \rangle \gamma'^\ell + \beta'^\ell}{\epsilon^\ell_{\mathbf{x}}} - \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1} \rangle \hat{\gamma}^\ell + \hat{\beta}^\ell \right) / 2^D . \tag{12}$$

[1]Without loss of generality, we can assume $\gamma'^\ell > 0$; indeed, if $\gamma'^\ell < 0$ it is sufficient to flip the signs of both $\mathbf{w}^\ell$ and γ'^ℓ.

To pinpoint the possible sources of errors, we expand this difference by a repeated application of the sum-and-subtract equation property:

$$
\begin{aligned}
&\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell + \beta'^\ell}{\epsilon_{\mathbf{x}}^\ell} - \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\hat{\gamma}^\ell + \hat{\beta}^\ell\right)/2^D = \\
&= \left[\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell + \beta'^\ell}{\epsilon_{\mathbf{x}}^\ell} - \left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right)\right] + \\
&\quad + \left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\hat{\gamma}^\ell + \hat{\beta}^\ell\right)/2^D = \\
&= \left[\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell + \beta'^\ell}{\epsilon_{\mathbf{x}}^\ell} - \left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right)\right] + \\
&\quad + \left[\left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\frac{\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right)\right] + \\
&\quad + \left(\frac{\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\hat{\gamma}^\ell + \hat{\beta}^\ell\right)/2^D = \\
&= \left[\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell + \beta'^\ell}{\epsilon_{\mathbf{x}}^\ell} - \left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right)\right] + \\
&\quad + \left[\left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\frac{\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right)\right] + \\
&\quad + \left[\left(\frac{\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\frac{\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right)\right] + \\
&\quad + \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\frac{\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\hat{\gamma}^\ell + \hat{\beta}^\ell\right)/2^D\,.
\end{aligned}
\tag{13}
$$

3 Methodology and experimental setup

This project will consist of two parts: a theoretical and experimental analysis of the discrepancy between a FQ neuron and its TQ counterpart, and an experimental investigation of the impact of fake-quantisation on the training of a realistic CNN.

3.1 FQ and TQ neurons

In this part of the project, we will analyse the error propagation involved in the conversion of a single FQ neuron map (9) into a TQ neuron map (10). At this stage, the complexity of the problem should by sufficiently low to allow us to perform some numerical analysis and establish error bounds [8].

We will pay particular attention to the errors arising from the second term of (13):

$$
\left(\frac{\langle \epsilon_{\mathbf{w}^\ell}\hat{\mathbf{w}}^\ell, \epsilon_{\mathbf{x}^{\ell-1}}\hat{\mathbf{x}}^{\ell-1}\rangle\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right) - \left(\frac{\langle \hat{\mathbf{w}}^\ell, \hat{\mathbf{x}}^{\ell-1}\rangle\epsilon_{\mathbf{w}^\ell}\epsilon_{\mathbf{x}^{\ell-1}}\gamma'^\ell}{\epsilon_{\mathbf{x}}^\ell} + \frac{\beta'^\ell}{\epsilon_{\mathbf{x}}^\ell}\right).
$$

Indeed, we hypothesise that the dot product between fake-quantised arrays is the operation where floating-point representations can create the largest discrepancies between FQ and TQ neuron maps.

3.2 MobileNetV1 on CIFAR-10

In this part of the project, we will assess the impact of fake-quantisation on the performance of a realistic CNN: a MobileNetV1 network topology [1] solving the CIFAR-10 image classification task [9].

We define the discrepancy between the FQ and TQ versions of a network as the collection of the distributions of absolute errors at each layer. We will measure this discrepancy for several configurations; the space of configurations is defined as the Cartesian product of three degrees of freedom:

- the target precision of weights and activations (in number of bits);
- the size of the weight filters (in terms of spatial dimensions and number of input channels);
- the floating-point format used to represent FQ operands.

We will explore several floating-point formats, both defined by the IEEE754-2008 standard [10] and the non-standard "brain floating-point" 16-bit format [11]:

- `FP32`, also known as *full precision*;
- `FP64`, also known as *double precision*;
- `FP16`, also known as *half precision*;
- `bfloat16`.

We might use existing open-source software to emulate different floating-point formats [12].

References

[1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: efficient convolutional neural networks for mobile vision applications," 2017.

[2] 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 Fourth International Conference on Learning Representations (ICLR 2016)*, International Conference on Learning Representations (ICLR), 2016.

[3] 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.

[4] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers," in *Proceedings of the Third Conference on Machine Learning and Systems (MLSys 2020)*, 2020.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 26th Conference on Neural Information Processing Systems (NIPS 2012)*, Neural Information Processing Systems (NIPS), 2012.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the Third International Conference on Learning Representations (ICLR 2015)*, International Conference on Learning Representations (ICLR), 2015.

[7] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, MLResearchPress, 2015.

[8] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, pp. 5–48, 1991.

[9] A. Krizhevsky, V. Nair, and G. E. Hinton. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2014.

[10] "IEEE Standard for Floating-Point Arithmetic," standard, The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, June 2008.

[11] S. Wang and P. Kanwar. `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`, 2019.

[12] S. Mach and G. Tagliavini. `https://github.com/oprecomp/flexfloat`, 2018.