Reduced/Mixed precision in Machine Learning

Shao Jie Hu Chen, Jaesung Kim

Abstract

Quantization techniques are methods applied to use low-precision datatypes to speed up inference time and reduce energy consumption for operations performed in a computer. In Machine Learning, they have been applied to different situations, such as Gaussian Processes, Kernel approximation, and (specially) Neural Networks. This paper provides an overview of the current quantization techniques applied in the mentioned situations, identifying the features, challenges, and benefits achieved. In Gaussian Processes, a low precision approach is used to efficiently compute the gradient for a Gaussian Process model. In Kernel approximation, a direct parameters quantization is performed, as well as an approximation to kernel using quantized values. In Neural Networks, general quantization techniques are explained (per channel/tensor quantization and quantization-aware training/post-training quantization), as well as specific structures. Quantization in neural network can be applied by mixing a low precision and full precision according to its role while training and inferring. That could narrow down into 4 bit level or even extremely binary stage. Although quantization has provided successful results in specific use cases (discussed in this paper), several challenges remain to be solved. Quantization ideas for Gaussian Processes and Kernel methods are still in premature state, and a general method for quantizing Neural Networks is still in development. Therefore, although promising, different challenges still remain for reduced/mixed precision in Machine Learning, as discussed in the conclusion.

Index Terms

Machine Learning Algorithms, Quantization, Low Precision, Mixed Precision.

I. Introduction

The numerous applications of Machine Learning in different fields have made it one of the main research fields in the scientific communities, with numerous groundbreaking results that have impacted in our daily life. Several examples may be found, such as the usage of large language model chatbots (like ChatGPT) which allows us to ask questions and receive answers based on the data it contains inside or the implementation of speech processing algorithms (like Google Assistant) that allows us to send instructions comfortably with some voice commands.

Although its contribution have undeniably changed our societies, the large resources that are needed to train these models are a great concern. The computational resources needed, along with the electricity consumption and hardware burnout needed, make the machine learning infrastructure become economically, technically, and environmentally unsustainable, as discussed in [1].

On the other hand, several applications of machine learning models in resource limited devices have been implemented in recent years, motivated by the potential of these models in edge computing. [2], [3], [4] Therefore, the enormous computations resources usually required by these models have to be simplified so that they can be run on these devices. [5], [6]

Therefore, a need for a more efficient way to use machine learning models have arised during recent years. Several ways have been made to do so, such as pruning neural networks. [7], [8] In this report, we discuss one approach to deal with this problem, which is the usage of low/mixed precision in Machine Learning, which we shall explain in details.

A. Concept of low/mixed precision in Machine Learning

Typically, for a given number of b bits used for each number, several number representation systems can be found in a computer. If our purpose is to represent natural numbers, the most typical one is to assign each number representable by the bits such as the lowest number is 0 and each number higher than that is increased by 1. For example, if 4 bits are used for number representation, $0000_2 = 0_{10}$, $0001_2 = 1_{10}$ and $0010_2 = 2_{10}$. On the other hand, if we want to represent integer numbers, the same philosophy apply, but in order to represent negative number too, we have to reserve first bit s to represent sign. By convention, we can establish that $s = 0$ represents positive numbers, whereas $s = 1$ represents negative numbers.

In constrast, if we want to represent real numbers, a different methodology has to be applied. The standard representation used in current computers is the floating-point representation system specified in IEEE-754. As represented in Figure 1, for $b = 32$ bits, the first bit is used to indicate sign (0 for positive numbers, 1 for negative numbers), the next 8 bits are used to represent exponent and the remaining bits are used to represent the fractional part of the value.

The precision and range that can be achieved for each of these datatypes varies, as well as the overheat in terms of time and computation resources needed to operate with them. In Table I, we can see a comparison of these datatypes, as well as other usual datatypes within a computer. In this table, we have not included a direct comparation of the speed-up of each operation (sum, multiplication and division) for the different datatypes, as these values strongly depend on the architecture and internal optimization of each CPU. Just for reference purposes, for an Intel Core i3 2370M, sum and multiplication are 38% faster in int32 than in float32.

Fig. 1. IEEE 754 representation using 32 bits. Source: Wikimedia Commons.

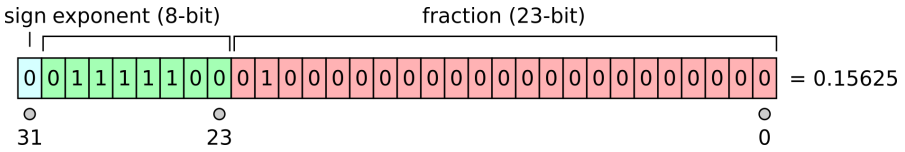

TABLE I
COMPARISON BETWEEN DIFFERENT DATATYPES REPRESENTATIONS. PRECISION IS ESTABLISHED ACCORDING TO WIDESPREAD DEFINITION IN DIFFERENT PROGRAMMING LANGUAGES.

Datatype	Description	Bits	Representation range	Precision
float32	Floating-point representation according to IEEE-754.	32	$[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$	1.19×10^{-7}
float16	Half precision floating-point representation according to IEEE-754.	16	$[-6.6 \times 10^{4}, 6.6 \times 10^{4}]$	4.88×10^{-4}
uint32	Natural numbers representation.	32	$[0, 4.3 \times 10^{6}]$	0.5
int32	Integer representation.	32	$[-2.15 \times 10^{6}, 2.15 \times 10^{6}]$	0.5
int16	Integer representation.	16	$[-3.2 \times 10^{4}, 3.2 \times 10^{4}]$	0.5
int8	Integer representation.	8	$[-128, 127]$	0.5
int4	Integer representation.	4	$[-8, 7]$	0.5
bool	Boolean representation.	1	true, false	-

Usually, the values used in Machine Learning require the precision of floating-point representation. For example, neural networks weights in Pytorch [9] and Tensorflow [10] are represented using 32 bits and 64 bits of floating-point representation, respectively. However, the high computational costs of the operations using this datatypes compared to other ones motivate the usage of other datatypes with **lower precision** than floating-point representation, or even using **mixed precision** by combining floating-point and other datatypes in the same model. However, as a consequence the results obtained using these datatypes are less precise than the ones obtained from full precision (specially in situations where thousands of operations are being done), so that is a point that needs to be considered in detailed if quantization is used, as will be noted.

B. Quantization in other fields

The usage of reduced precision has not been exclusively thought for Machine Learning. In several other fields, reducing the precision from initial values (typically when continuous values/functions need to be represented in a computer) has been useful. Different examples follow:

- **Signal processing**. Typically, a signal is a continuous function, whose amplitude indicate the amplitude of air perturbation or amplitude of electromagnetic waves. Their implementation in computers is highly important, specially for communication between different devices. As in a computer the representation of continuous values is difficult, a discretization scheme has been used to transform each signal in different magnitudes, so that they could be easily treated inside a computer.
- **Information theory**. A usual technique used for reducing the memory required to store an image/video is to apply data compression techniques. An example can be found in JPEG, in which quantization matrices are used in order to reduce the memory footprint of the compressed image.
- **Numerical methods**. Recent works have used quantization schemes in order to approximate numerical solutions to partial differential equations. In particular, low precision has been applied in [11] to approximate solutions to the heat equation with bounded error rate.

C. Challenges in low and mixed precision in Machine Learning

Implementing reduced precision methods in Machine Learning may lead to several advantages, as explained in [12]. On the one hand, quantized Machine Learning models allow to decrease the consumption of time and energy, which is very beneficial for edge computing devices. On the other hand, the memory overheat required to run these models is reduced significantly with respect to using full precision models, as shall be noted in the examples in the following sections. Nevertheless, different challenges arise from their study. Namely, the main considerations that need to be taken into account are:

1) **Choose a correct representation scale using lower precision datatypes.** When dealing with a specific use case, the trade-off between accuracy and overheat has to be considered when choosing a datatype. In Table I we may observe the representation range and the precision achieved by each datatype. A correct choice of representation scale is essential for correct computation.
2) **Choose the quantization method used for each use case.** The manner in which the datatypes are quantized may also affect to the final computation. Dividing the range of values in equal chunks will suffice? Maybe for some specific use cases, it is more interesting to concentrate values in intervals where there is a concentration of values.

3) **Study the stability of the results/predictions obtained in a low/mixed precision Machine Learning model with respect to full precision version.** As after quantization we inevitably lose precision in each computation, a special consideration has to be for stability of the final solution. Will minor changes in precision in individual operations affect to the final prediction?
4) **Study the variation of precision of low/mixed precision Machine Learning model with respect to full precision version.** In relation with the previous point, can the variation of precision in final prediction be controlled/bounded by some theoretical results? Under which circumstances can we bound the prediction error induced by quantization?

In the following sections, these topics shall be discussed in details. Section II explains scales used to quantize values in Machine Learning models, as well as techniques used to reduce the loss of information caused by quantization. In Sections III, IV and V, low/mixed precision is discussed in different models/algorithms in Machine Learning, outlining which challenges were achieved in each situation and providing empirical results. Section VI concludes the report and provides future lines of research in quantization for Machine Learning.

II. Quantization scales

For a Machine Learning application where the operations are performed using float32/float64 datatypes (full precision), we can change the representation of these numbers to another format, as the ones shown in Table I. Several techniques are used in the state-of-the-art, which can be divided into the 2 following categories: (i) uniform quantization, and (ii) non-uniform quantization.

A. Uniform quantization

In this technique, all the data are mapped to the range of the reduced quantized values such that the difference between each two consecutive values remains the same.

As a motivation to define the transformation, we first consider an example with a fixed interval $I = [a, b]$ (which contains all of the data we have) to a new range $\tilde{I} = [\tilde{a}, \tilde{b}]$. In order to do an uniform transformation between this interval to another scale, we define an affine transformation $f : I \to \tilde{I}$ which maps the interval I into the new range, that is, $f(x) = ax + b$, with $f(a) = \tilde{a}$ and $f(b) = \tilde{b}$. If, instead of an interval $\tilde{I}$ the image were only discrete values, we just need to readjust the formula, so that the image is only discrete values. One way to go is to approximate ax and b by the lower bound integer, that is, considering the function $Q(x) = \lfloor x/S \rfloor - Z$, where we just have renamed the variables $a = \frac{1}{S}$ and $b = -Z$ and performed the approximation term by term.

Notice that, in the previous example, both intervals $I = [a, b]$ and $\tilde{I} = [\tilde{a}, \tilde{b}]$ may be ignored (and, in fact, while working with numbers in Machine Learning, the interval $I = [a, b]$ is usually undetermined) if we choose beforehand the values S and Z (which can be intuitively interpreted as a scaling term and a zero-point term, respectively). This motivates the following definition, which is the usual operation used to perform quantization.

Definition II.1 (Uniform quantization function)**.** Let $S \in \mathbb{R}$ and $Z \in \mathbb{N}$. A **quantization function** Q of scale S and zero-valued in Z is the mapping $Q : \mathbb{R} \to \mathbb{N}$, where:

$$Q(x) = \lfloor x/S \rfloor - Z, \forall x \in \mathbb{R} \tag{1}$$

The parameter S is called the **scaling factor** and the parameter Z is the **zero value factor**.

Comparably, we could define another function that, from a initial set of quantized values, it goes back to the original interval. We just need to define a function similar to the function defined in Equation 1, but without considering the approximation by the lower bound. This motivates the following definition:

Definition II.2 (Uniform dequantization function)**.** Let $S \in \mathbb{R}$ and $Z \in \mathbb{N}$ be, respectively, the scaling factor and the zero value factor. A **dequantization function** $\overline{Q}$ of scale S and zero-valued in Z is the mapping $\overline{Q} : \mathbb{N} \to \mathbb{R}$, where:

$$\overline{Q}(n) = S(n + Z), \forall n \in \mathbb{N} \tag{2}$$

As an important note, usually for a real value $x \in \mathbb{R}$, $\overline{Q}(Q(x)) \neq x$ (which means that $\overline{Q}$ is not the inverse function of Q). The main reason of this is because of the floor function used in Equation 1. The moral of this notice is that, once you apply a quantization function Q to an initial set of data, information is lost in the process and can't recovered afterward.

It is also important to notice that it is essential to choose the term scale factor S carefully. That is, because when it comes to the implementation of quantization in real-world devices, usually the range of values is fixed beforehand (by the number of bits used for the quantization). Therefore, if S is too small compared to the values, several numbers won't be represented adequately in the quantized system (there would be overflow). On the other hand, if S is too big, different values would have the same quantized value, leading to poor precision. Therefore, a correct adjustment to the term S is important. This process is called **calibration**. Several calibrations techniques have been used in the state-of-the-art, enumerated as follow:

1) **Min-max technique**. It consists of setting the lowest and highest values representable by the quantization schedule to map to the lowest $r_{\min}$ and highest value $r_{\max}$ in the real values to be considered, respectively. This way, all the values would have its corresponding number without needing to clip the values.
2) **Symmetric quantization technique**. This consists of choosing S such that, if $r_{\min}$ is the lowest value and $r_{\max}$ is the highest value, the parameter S is chosen such that the lowest value and the highest value are $\max(|r_{\min}|, |r_{\max}|)$.
3) **KL Divergence entropy**. This is the approach used in TensorRT, the machine learning framework (designed by NVIDIA for its own GPUs) to run Machine Learning models. The term S is chosen so that the original distribution and the quantized distribution by S minimize the Kullback–Leibler divergence.
4) **Percentile**. This consists of choosing S such as at least $x\%$ of values of the original distribution are correctly represented in the distribution (typically, it's the 99%).

B. Non-uniform quantization

The previous quantization scale is the simplest way to perform quantization. However, other quantization scales may be used. For example, instead of distributing equally the range of values to be quantized, we can opt to concentrate values to intervals where we know that provides more useful insight than other ranges. For examples, if in a model only a few outlines present values outside the interval $[-\Delta, \Delta]$, we could assign only a few values outside this interval and give more quantized values to data within this interval. This motivates the creation of non-uniform quantization functions.

Definition II.3 (Non-uniform quantization function)**.** Let $A = \{a_0, \ldots, a_m\}$ be the different quantized values and consider a partition of $\mathbb{R}$ $P = \{-\infty \leq \Delta_0 < \Delta_1 < \cdots < \Delta_{m+1} \leq \infty\}$. A **non-uniform quantization function** is the mapping $Q : \mathbb{R} \to A$, where:

$$Q(x) = a_i, \forall x \in [\Delta_i, \Delta_{i+1}), i \in \mathbb{N} \tag{3}$$

Intuitively, what we are doing with the expression from Equation 3 is represent each value from the interval $[\Delta_i, \Delta_{i+1})$ with a quantized value. Strictly attaining to the definition, an uniform quantization could also be considered as a special case of non-uniform quantization function. However, usually when the transformation is given by the Equation 1, uniform quantization may be assumed without confusion.

An example of non-uniform quantization is using a logarithmic quantized scale to the output, that is, $Q(x) = \lfloor \log_2(x/S) \rfloor - Z, \forall x \in \mathbb{R}$. This is the most usual non-uniform quantization function that can be found in the state of the art. [To be done: cite papers where logarithmic scale is used.]

As in the case of uniform quantization, a dequantization function can also be defined in these situations:

Definition II.4 (Non-uniform dequantization function)**.** Let $A = \{a_0, \ldots, a_m\}$ be the different quantized values associated to the partition of $\mathbb{R}$ $P = \{-\infty \leq \Delta_0 < \Delta_1 < \cdots < \Delta_{m+1} \leq \infty\}$. A **non-uniform dequantization function** is the mapping $\overline{Q} : A \to \mathbb{R}$ given by:

$$\overline{Q}(a_i) = \Delta_i, \forall i \in \{1, \ldots, m\} \tag{4}$$

Therefore, what we are doing now is, to each quantized value that represents each interval $[\Delta_i, \Delta_{i+1})$, we are assigning to the lowest value of the interval, Δ_i. Notice that different dequantization methods could have been done (for example, assigning the middle of the interval instead of the lowest value), but the approach presented here is the usual one chosen in the literature (for example, in one of the sections this logarithmic quantization is used). Also notice that Q and $\overline{Q}$ are not inverse one of another, as in the case of uniform quantization.

III. Low/mixed precision in Gaussian Processes

One Machine Learning algorithm that may be used are Gaussian Processes. In this section, consider a given dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, where $x_i \in \mathbb{R}^D$ and $y_i \in \mathbb{R}$ (D is the number of features to be considered). We want to use a Gaussian Process model to approximate the values:

$$y_i = f(x_i) + \epsilon_i, \text{ where } f(.) \sim \mathcal{GP}(m(.), k(.,.)), \epsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{5}$$

In Equation 5, $m(.)$, $k(.,.)$ are the mean function and the covariance kernel, respectively. Without loss of generality, we may assume that the mean function is identically zero. [13] The negative log marginal likelihood is given in this case by the following expression:

$$\mathcal{L}(\theta) = -\log p(y|X, \theta) = \frac{1}{2} log(|\tilde{K}|) + \frac{1}{2} y^T \tilde{K}^{-1} + \frac{N}{2} \log(2\pi), \text{ where } \tilde{K} = (K + \sigma^2 I)_{i,j} \tag{6}$$

From Equation 6, the gradient must be derived to estimate the hyperparameters θ:

$$\nabla_\theta \mathcal{L} = \frac{1}{2}\,\mathrm{Tr}\left(\tilde{K}^{-1}\nabla_\theta \tilde{K}\right) - \frac{1}{2}y^T\tilde{K}^{-1}\left(\nabla_\theta \tilde{K}\right)\tilde{K}^{-1}y \approx \frac{1}{M}\sum_{i=1}^{M} z_i^\top \tilde{K}^{-1}\nabla_\theta \tilde{K} z_i) - \frac{1}{2}y^T\tilde{K}^{-1}\left(\nabla_\theta \tilde{K}\right)\tilde{K}^{-1}y \tag{7}$$

For Equation 7, the computation of $\tilde{K}^{-1}y$ has a complexity of $\mathcal{O}(N^3)$ if direct computation using Cholesky decomposition is applied, so it's ineffective for use cases where N is large. Furthermore, in general the Cholesky decomposition is subject to numerical inestability, so application of reduced precision is not suitable in this case.

However, low precision can still be applied to Gaussian Processes, as done in [14]. To do so, they propose using a numerically stable iterative method to solve this linear equation, called **conjugate gradient method**. This method allows to simplify the computation for $\tilde{K}^{-1}y$ by resolving the equation system $\tilde{K}x = y$, reducing the complexity from $\mathcal{O}(N^3)$ to $\mathcal{O}(JN^2)$, where J is the number of iteration steps. As usual in iterative numerical methods, we start from an initial estimation x_0 and operate from this to obtain a succession of solutions $(x_0, \cdots, x_J)$ that converges to the solution of the system. The algorithm is described in Algorithm 1.

Algorithm 1 Conjugate Gradient algorithm. Derived from [14].

Require: Linear system $\tilde{K}x = y$. Initial estimation x_0, tolerance ϵ, preconditioner function $P(.)$
Ensure: An estimated solution x_J, $J \in \mathbb{N}$, such that $x_J \approx \tilde{K}^{-1}y$.
1: Initialize values: $k = 0$, $r_0 = \tilde{K}(x_0) - y$, $d_0 = -r_0$, $z_0 = P(r_0)$ and $\gamma_0 = r_0^T z_0$.
2: **while** $\|r_k\|_2 > \epsilon$ **do**
3: Calculate next iteration: $x_{k+1} = x_k + \alpha_k d_k$, where $\alpha_k = \frac{\gamma_k}{d_k^T \tilde{K} d_k}$.
4: Prepare next iteration (1): $r_{k+1} = r_k - \alpha_k \tilde{K} d_k$, $z_{k+1} = P(r_{k+1})$, $\gamma_{k+1} = r_{k+1}^T z_{k+1}$.
5: Prepare next iteration (2): $\beta_{k+1} = \frac{\gamma_{k+1}}{\gamma_k}$ and $d_{k+1} = -r_{k+1} + \beta_{k+1} d_k$.
6: Increase k by 1.
7: **end while**

This original algorithm may be quantized by reducing the precision to half precision in the term γ_n. However, the authors in [14] did not achieve good results after learning using a direct quantization to float16. Even though the conjugate gradient method is more numerically stable than Cholesky decomposition, there were still some inestability which needed to be solved. Therefore, the authors enhanced the stability by using Algorithm 2. In order to do so, the following ideas were implemented:

1) **Conjugate gradient rescaling**. As overflow error was detected initially, the authors saved the values of γ_n using a logarithmic scale. In order to compute $\log w^T z$, where w and z are a vectors, the transformation $\mathrm{L\Sigma E}\left(\mathrm{w}^T\mathrm{z}\right) = y_{\max} + \log\left(\sum_{i=1}^{N} s_i \exp\left(y_i - y_{\max}\right)\right)$ was performed.
2) **Re-orthogonalization**. In theory, the residual vector r_k should be orthogonal between them. However, this property may be lost when using reduced precision. Therefore, the authors proposed a re-orthogonalization at each step in order to specifically maintain this property.

Algorithm 2 Enhanced Stability Conjugate Gradient algorithm. Derived from [14].

Require: Linear system $\tilde{K}x = y$. Initial estimation x_0, tolerance ϵ, preconditioner function $P(.)$
Ensure: An estimated solution x_J, $J \in \mathbb{N}$, such that $x_J \approx \tilde{K}^{-1}y$.
1: Initialize values: $k = 0$, $r_0 = \tilde{K}(x_0) - y$, $d_0 = -r_0$, $z_0 = P(r_0)$ and $\log \gamma_0 = L\Sigma E(r_0^T z_0)$.
2: **while** $\|r_k\|_2 > \epsilon$ **do**
3: Calculate next iteration: $x_{k+1} = x_k + \alpha_k d_k$, where $\alpha_k = \exp\left(\log \gamma_k - \mathrm{L}\left(d_k^T K d_k\right)\right)$.
4: Prepare next iteration (1): $r_{k+1} = r_k - \alpha_k \tilde{K} d_k$, $z_{k+1} = P(r_{k+1})$, $\log \gamma_{k+1} = \mathrm{L\Sigma E}\left(\mathrm{r}_{k+1}^T z_{k+1}\right)$.
5: Re-orthogonalization of r_{k+1}: $r_{k+1} = r_{k+1} - \sum_{j=0}^{k} u_j^T r_{k+1} u_j$.
6: Prepare next iteration (2): $\beta_{k+1} = \exp\left(\log \gamma_{k+1} - \log \gamma_k\right)$ and $d_{k+1} = -r_{k+1} + \beta_{k+1} d_k$.
7: Increase k by 1.
8: **end while**

From this method developed by the authors of [14], the study of the convergence of the result could be interesting. Unfortunately, a general convergence result was not established in their original work (and, to our knowledge, it has not been discussed in-depth yet).

However, the authors indicate a weaker result which could help to discuss the convergence of the method with respect to full precision. For generalization bounds of Gaussian Processes methods, an important metric is the **effective dimension of a kernel** K, defined as the value $N_{\text{eff}}(K, \sigma^2) = \sum_{i=1}^{N} \frac{\lambda_i}{\lambda_i + \sigma^2}$, where λ_i represents the eigenvalues of K and σ^2 is a hyperparameter called noise. As proved in [15], the lower the value of N_{eff}, the better the generalization bounds. The authors in [14] proved the following result for their quantized model:

Theorem III.1 (Bounded generalization of Enhanced Stability Conjugate Gradient, adapted from [14])**.** . Let K be the covariance kernel of a Gaussian Process model, $(\lambda_i)_{i\in I}$ be the eigenvalues of K, σ^2 the noise parameter and Q be a quantization function. Then, the following inequality holds:

$$\mathbb{E}\left(\sum_{i=1}^{N}\frac{Q(\lambda_i)}{Q(\lambda_i)+\sigma^2}\right) \geq N_{\text{eff}}(K,\sigma^2) \tag{8}$$

Intuitively, the Equation 8 indicates that the effective dimension for a quantized function is expected to perform worse in terms of generalization than the full precision version (as expected).

On the benchmark experiment comparing single precision and half precision Gaussian Process, it showed speedups of at least two times over single precision especially on large dataset. While standard half precision conjugate gradients fails to converge, stable half precision conjugate gradient matched closely to convergence behavior of single precision ones. Through the experiment, however, lower precision is seemed to slow the rate of decay of the eigenspectrum of the kernel matrices. By using lower precisions, it deteriorated both on the training convergence, by slowing the rate of progress of the conjugate gradient iterations, and on the provable generalization guarantees by increasing the effective dimensionality.

IV. Low/mixed precision in Kernel approximation

One interesting application of the quantization methods in Machine Learning can be found in Kernel approximation. For high dimensions features, the usage of Kernel methods (family of ML algorithms which rely on a kernel matrix K to map an initial input data space into a higher dimensional feature space) becomes computationally expensive when the number of parameters increases. Different ideas have been used in the state-of-the-art to apply quantization techniques in Kernel methods. We can highlight two different approaches: (i) parameters quantizations and (ii) Random Fourier Features.

A. Parameters quantizations

One way to approach this problem is to quantize the different values of the kernel matrix k, as done in [16]. The formal definition of quantized kernel may be found in Definition IV.1.

Definition IV.1 (Quantized kernel)**.** Let $K \in \mathbb{R}^{N\times N}$ be a kernel. A **quantized kernel** is a mapping $k_q : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ such that there exists a function $q : \mathbb{R}^D \to \mathbb{N}^D$, with $q(x) = (q_1(x_1), \cdots, q_D(x_D))$, with $q_i : \mathbb{R} \to \mathbb{N}$ a quantization function for all $i \in \{1, \cdots, D\}$, such that:

$$k_q(x, y) = K(q(x), q(y)), \forall x, y \in \mathbb{R}^D \tag{9}$$

Intuitively, a quantized kernel k_q is assigning each pair (x, y) to the same values of original kernel K, but where the input was quantized beforehand. The main interest of quantization in Kernel methods may be found in use cases where high dimensionality is found (as stated before, Kernel methods become too computationally expensive when the number of parameters increases). The quantized kernel may be built using the following techniques:

1) **Additive quantized kernel**. The main idea behind this technique is to apply quantization term by term and, supposing that data dimensions are independent, the value of each kernel is also calculated term by term. Mathematically:

$$k_q(x, y) = \sum_{d=1}^{D} k_{q_d}(x_d, y_d) = \sum_{d=1}^{D} \varphi_{q_d}(x_d)^\top K_d \varphi_{q_d}(y_d) \tag{10}$$

   In other words, it's about applying quantization term by term and supposing that the kernel K is "diagonal". As may be nosted, the disadvantage of this technique is that independence in the data dimensions has to be assumed for each of the elements of the dataset (which rarely happens in practice).
2) **Block quantized kernels**. This idea generalizes the previous one. Instead of considering each dimension separately, a quantization by arranging a set of dimensions is performed (the kernel K is a "block matrix"). This way, the hypothesis of independence between the different dimensions may be weakened by supposing independence for groups of dimensions.

The previous techniques were used in [16] for feature matching application only, but these ideas could be generalized to apply for any use case where kernel methods are involved. However, to our knowledge, no further research in this topic has been made so far. An interesting approach could be studying the convergence properties of these concepts with respect to full precision approach.

Additionally, the authors of [16] showed an experiment on the visual feature matching benchmark, which is a scheme for learning discriminative, low dimensional image descriptors from realistic training data. The matching performance of quantized kernel compared to ordinary kernel, for example Euclidean distance or X^2 kernel, outperformed exceedingly with just a few bits to represent each feature dimension. With respect to the most compact encodings ($\leq$ 64bits), after pre-processing the data with PCA (a method for dimensional reduction) projecting to 32 dimension and then using quantized kernel, it brought better performance. This is because of potential benefits of decorrelating features and joint compression. In this regard, the author propose the future exploration on higher compression ratio of feature dimensions.

B. Random Fourier Features

Another way to address this problem is to use Random Fourier Features, which allow to efficiently approximate the kernel. The usage of Random Fourier Features can be computed even faster if low precision is used, as explored in [17].

In order to do so, consider a generic kernel matrix $K \in \mathbb{R}^{N\times N}$ matrix, where $N \in \mathbb{N}$ is large. We want to approximate each element of the kernel with a randomized mapping $z : \mathbb{R}^D \to \mathbb{R}^R$, where D is the dimension of each feature of the dataset considered (ideally $R << N$), such that $k(x, y) \approx z(x)^T z(y)$. This may be done by using the Random Fourier Features, where each component of the vector $z(x) = [z_1(x), z_2(x), \ldots, z_D(x)]^T$ (where x is a new data point to be predicted) is calculated using Algorithm 3.

Algorithm 3 Random Fourier Features (RFF) computation. Obtained from [18].

Require: A kernel function such that $k(x, y) = k(x - y)$.
Ensure: Randomized feature map $z : \mathbb{R}^D \to \mathbb{R}^N$ which approximates the kernel function $k(x, y) \approx z(x)^T z(y)$.
1: Compute Fourier transform p of kernel: $p(\omega) = \frac{1}{2\pi}\int e^{-i\omega^T\delta}k(\delta)d\delta$
2: Draw $\omega_1, \ldots, \omega_N \in \mathbb{R}^D$ samples iid from Fourier Transform p. Draw $b_1, \ldots, b_N \in \mathbb{R}$ iid from uniform distribution on $[0, 2\pi]$.
3: Define $z(x) = \sqrt{\frac{2}{D}}[\cos(\omega_1^T x + b_1) \ldots \cos(\omega_N^T x + b_N)]^T$.

The authors in [17] propose to quantize the values obtained using the Random Fourier Features. This is done by following the Algorithm 4. In summary, for a given new point x, the algorithm quantize each of each feature $z_i(x), i \in D$ obtained in Algorithm 3, rounding them randomly to either top or bottom of the interval such as the expected value is $z_i(x)$. This allow to store each data in b bits, with b being smaller than the usual amount of bits used in full-precision (usually, 32 bits).

Algorithm 4 Low-Precision Random Fourier Features (LP-RFF) computation.

Require: Feature map $z : \mathbb{R}^D \to \mathbb{R}^N$ obtained from Algorithm 3. A value $x \in \mathbb{R}^D$.
Ensure: Quantized value $\tilde{z}(x)$.
1: Initialize $\tilde{z}(x)$.
2: **for** $i \in \{1, \ldots, N\}$ **do**
3: Calculate $z = z_i(x)$.
4: Round z to $\overline{z}$ with probability $\frac{\overline{z}-z}{\overline{z}-\underline{z}}$ and to $\underline{z}$ with probability $\frac{z-\underline{z}}{\overline{z}-\underline{z}}$
5: Store in $\tilde{z}_i(x)$ the integer $j \in \mathbb{Z}$ corresponding to the rounded value of z.
6: **end for**

From a theoretical point of view, we have to prove that both approaches approximate correctly to the kernel function k. Several theoretical metrics can be done for this task. For the comparison between Random Fourier Features and Low Precision Random Fourier Features, the Δ-spectral approximation, or, its variant, the (Δ_1, Δ_2)-spectral approximation is used.

Definition IV.2 ((Δ_1, Δ_2)-spectral approximation, [19])**.** Let A, B be symmetric matrices with the same order. For $\Delta_1, \Delta_2 \in \mathbb{R}^+$, A is a (Δ_1, Δ_2)-spectral approximation of B if $(1 - \Delta_1)B \preceq A \preceq (1 + \Delta_2)B$ (where $X \preceq Y \iff Y - X$ is positive semidefinite matrix).

For the full precision version, a result of convergence of the method was proved in [17].

Theorem IV.1 (Convergence of Random Fourier Features, [17])**.** Let K be a kernel and $\tilde{K}$ be an approximation of K using Random Fourier Features as indicated in Algorithm 3. Suppose that $\tilde{K} + \lambda I$ is a (Δ_1, Δ_2)-spectral approximation of $K + \lambda I$, with $\Delta_1 \in [0, 1)$, $\Delta_2 \geq 0$ and $\lambda \geq 0$ a regularization constant. The following inequality holds:

$$\mathcal{R}(f_{\tilde{K}}) \leq \frac{1}{1-\Delta_1}\hat{\mathcal{R}}(f_K) + \frac{\Delta_2}{1+\Delta_2}k_m \tag{11}$$

$\mathcal{R}(f_{\tilde{K}})$ denotes the expected error for the approximated kernel $\tilde{K}$, $\hat{\mathcal{R}}(f_K)$ is an upper bound for the estimated error for the kernel K and k_M is a constant that depends on $m = rank(\tilde{K})$.

From this result, we can see the relevance of (Δ_1, Δ_2)-spectral approximation for our purpose. As Δ_1 and Δ_2 become smaller, the better the approximation performed by $\tilde{K}$. This result is significantly relevant for our case. It says that the approximation $\tilde{K}$ of a kernel K using Random Fourier Features is good as long as $\tilde{K} + \lambda I$ is a (Δ_1, Δ_2)-spectral approximation of $K + \lambda I$.

On the other hand, in [17] a result for low precision was also proved:

Theorem IV.2 (Convergence of Low Precision Random Fourier Features, [17])**.** Let K be a kernel, $\tilde{K}$ be an approximation of K using Low Precision Random Fourier Feature using b bits as indicated in Algorithm 4 and $\lambda \geq 0$ a regularization constant. Suppose that $\|K\| \geq \lambda \geq \delta_b^2 = 2/(2^b - 1)^2$. If $\Delta_1 \geq 0$ and $\Delta_2 \geq \delta_b^2/\lambda$, then the following inequality holds:

$$P[(1-\Delta_1)(K+\lambda I_n) \preceq \tilde{K}+\lambda I_n \preceq (1+\Delta_2)(K+\lambda I_n)] \geq 1-a\left(\exp\left(b_m\right)+\exp\left(c_m\right)\right) \tag{12}$$

where $a = 8\,\mathrm{tr}\left((K+\lambda I_n)^{-1}(K+\delta_b^2 I_n)\right)$, $b_m = \frac{-m\Delta_1^2}{\frac{4n}{\lambda}(1+\frac{2}{3}\Delta_1)}$ and $c_m = \frac{-m(\Delta_2-\frac{\delta_b^2}{\lambda})^2}{\frac{4n}{\lambda}(1+\frac{2}{3}(\Delta_2-\frac{\delta_b^2}{\lambda}))}$.

As a consequence, by denoting $1-\rho = 1-a\left(\exp\left(b_m\right)+\exp\left(c_m\right)\right)$:

- If $\Delta_1 \leq 3/2$ and $m \geq \frac{8n/\lambda}{\Delta_1^2}\log\left(\frac{a}{\rho}\right)$, then $(1-\Delta_1)(K+\lambda I_n) \preceq \tilde{K}+\lambda I_n$ with probability at least $1-\rho$.
- If $\Delta_2 \in [\delta_b^2/\lambda, 3/2]$ and $m \geq \frac{8n/\lambda}{(\Delta_2-\delta_b^2/\lambda)^2}\log\left(\frac{a}{\rho}\right)$, then $\tilde{K}+\lambda I_n \preceq (1+\Delta_2)(K+\lambda I_n)$ with probability at least $1-\rho$.

The Theorem IV.2 states that as long as we have a good (Δ_1, Δ_2)-spectral approximation of kernel K using Low Precision Random Fourier Feature, a good estimation may be achieved even using b bits of precision (as long as the loss of precision respects the assumptions used in the indicated theorem).

The authors in [17] compared the trade-off between accuracy and memory obtained between Low Precision Random Fourier Features and (full precision) Random Fourier Features by running both versions in different datasets. Figure 2 shows this comparation for CovType dataset for a good (Δ_1, Δ_2)-approximation of the kernel. [20] The objective of the model in this dataset is to correctly classify the cover type of forests based on physical variables. As it may be observed, the low precision version of Random Fourier Features achieves similar results in terms of accuracy with respect to full precision algorithm. In addition, given a restriction of available memory, low precision version tends to have a better result than full precision version. In the best case, a compression of x4.7 in memory size is achieved for this dataset (see Table 2 in [17]). Unfortunately, this paper does not refer to inference time differences between these 2 approaches, but it certainly provides evidence of how useful low precision can be in Machine Learning when memory size is restricted.

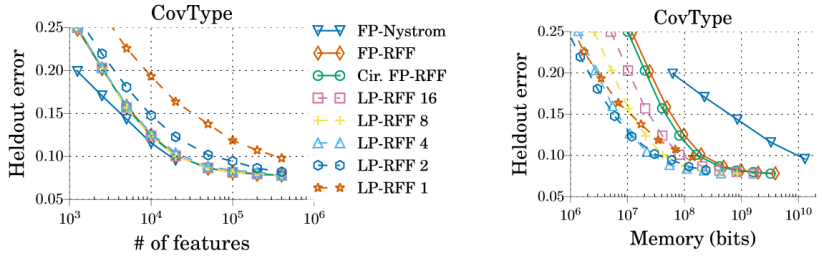

Fig. 2. Comparison of error obtained in CovType dataset [20] between Low Precision Random Fourier Features and and (full precision) Random Fourier Features with respect to the number of features and the memory used, respectively. Each curve is named following the format PP-TTT (N). PP refers to floating point (FP) or low precision (LP). TTT refers to the technique used: Random Fourier Features (RFF) or Nystrom's method, which is another method used to calculate kernel K. N indicates the number of bits used for low precision. Obtained from [17].

V. Low/mixed precision in Neural Networks

ML algorithms based on artificial neural networks are one of the most popular and studied algorithms in the state of the art. Inspired by the nature of neural networks in Neuroscience, these algorithms try to learn complex information by using a network of minimal connection units: neurons.

An artificial neural network may be characterized by the weights w, biases b and activations ϕ defined for each neuron in the network (typically, the neurons are organized in layers so that the each layer outputs their values to the next layer or to itself). By correctly finetuning the parameters within an artificial neural network, we may obtain a powerful ML model. However, the number of parameters is huge in actual use cases and operating using full precision may be too excessive. For example, the VGG-16 [21] network contains about 140 million 32-bit floating parameters and recent CNN models contain even more. Quantization methods have been extensively studied for artificial neural networks because of its potential benefits in terms of memory-saving and inference speed-up.

A. Quantization granularity and training

The weights w and the biases b may be quantized using the schemes presented in Section II. When the number of parameters is large, in may be possible that the same quantization scheme may not be globally optimal (for example, if the range of values in parameters w and b is too large, there could be even neurons where all of its values are zeros in the worst case!).

This problem is solved by quantizing the parameters differently for each layer. Two typical schemes are used, depending whether the same quantization is applied to the whole layer (**layerwise quantization**) or applied differently for subgroups of neurons in the layer (**groupwise quantization**). [22] Generally, the latter offers better performance (as it allows to represent more accurately the range of values of each subgroup instead of quantizing the whole layer), but at the expense of consuming more memory to store the quantization scheme used.

In general, once we have a neural network with its weights w and bias b, we quantize these parameters in order to obtained a quantized version of the neural network. However, due to the change of parameters value, maybe the network's output may be significantly different to the result outputted from the original one. Therefore, sometimes retraining of the quantized parameters is required in order to fine-tune the quantized neural network. The process of just quantizing the parameters after training is called **Post-Training Quantization**, whereas the process of re-training the model taking into account the quantization scheme used is called **Quantization-Aware Training**. [22] In general, the quantized neural network produces better results if Quantization-Aware-Training was applied, but at the expense of using more computational resources in order to retrain the parameters.

B. Mixed precision

The core idea behind mixed precision [23] is to balance the trade-off between precision and efficiency during numerical computations. In single precision, it typically leads to more accurate results but demands increased computational resources and memory usage due to the larger bit representation. On the other hand, lower precision reduces resource requirements but may compromise the quality or accuracy. Mixed precision addresses this challenge by strategically employing different precision levels for various components of a computation. It involves performing some calculations in lower precision but still maintaining a single-precision copy while training.

After describing the basic logic of mixed precision, efficient approaches to leverage mixed precision for neural networks will be discussed.

First, during the whole mixed precision the storage called FP32 master copy is implemented. While training, weights, activations and gradients are all represented as FP16. So before the weights get into forward propagation, weights from FP32 master copy are halved into FP16 and after the back propagation is finished, FP16 weight gradients are used to update weights from FP32 master copy. Using FP16 reduces huge amount of memory required for training, but in a view of update, it means the loss of gradient information. As gradients keep accumulated, it is important to retain gradient with meaningful value. However the range of what FP16 can represent is limited compared to FP32 and according to gradient distribution that was used for experiment in [24], 5% of weight gradient become zero as their exponent value exceeds the limit (2^{-24}) which FP16 can fully represent. Though maintaining extra weight storage requires 50% more memory capacity compared to not using it, consequently as the most of activation is held by 16 bit sizes, the overall memory consumption becomes roughly halved. Second, from the experiment of [23] that relevant gradient values go zero as they can't be represented with FP16, scaling the loss value computed in the forward pass before back propagation is introduced. On the experiment, Multibox SSD network [24] was trained and it shows only less than 30% was able to be representable with FP16 among the whole gradient values. By setting scale factor in 8, which means increasing the exponents of every gradient by 3, values in [2^{-27}, 2^{-24}) were able to be preserved therefore achieving similar accuracy with FP32 training.

As the above idea was preliminary applied in a layer, on complicated convolution network, mixed precision can be applied in every single layer even with multiple precision choices. However carrying a constant precision level at every layer won't be favored because quantizing different layers can have different impact on the accuracy and efficiency through overall network. Then the problem extends to searching optimal architecture among N layers and M candidate precisions. Brute force search for those combination would consume exhaustive time complexity ($\mathcal{O}(\mathcal{M}^{\mathcal{N}})$. To find the optimal architecture from complicated Conv layers, neural architecture search (NAS) is commonly used for efficiency. On NAS, search space defines the candidate operation (convolution, fully-connected, pooling etc.) and their combination presenting valid network configurations. According to search space and its goal, an appropriate search strategy needs to be designed not only covering the entire search space (exploration) but also catching the optimal from search space (exploitation). The optimal neural architecture from NAS problem can be established as

$$\min_{a \in A} \min_{w_a} \mathcal{L}(a, w_a) \tag{13}$$

where a denotes a neural architecture, A denotes the architecture space, and w_a denotes the weights of a.

[25] proposes a novel NAS approach where search space is defined in a stochastic manner with nodes representing data tensors and edges representing convolution operators with different bit-widths. Edges are executed stochastically and the probability of execution is parameterized by architecture parameter θ. This parameter determines which precision to adapt according to its probability distribution P_θ for both weights and activation. Search space can be presented as computational DAG (directed acyclic graph) denoted as $G = (V, E)$. Between two nodes v_i and v_j, edges connect them which are denoted as e_k^{ij} and they represent a convolution operator whose weights or activation are quantized to a lower precision. In other words each edge represents an operator parameterized by its trainable weight w_k^{ij}. For input node v_i, output node v_j is calculated summing the output of all incoming edges :

$$v_j = \sum_{i,k} (m_k^{ij} e_k^{ij}(v_i; w_k^{ij})) \tag{14}$$

where $m_k^{ij} \in 0, 1$ is an "edge-make" and $\sum_k (m_k^{ij}) = 1$ which works as a random variable of θ probability distribution. Each edge is assigned to a parameter θ_k^{ij} that determines the probability of executing e_k^{ij}

$$P_{\theta^{ij}}(m_k^{ij}=1) = softmax(\theta_k^{ij}|\theta^{ij}) = \frac{exp(\theta_k^{ij})}{\sum_{k=1}^{K^{ij}}(exp(\theta_k^{ij})} \tag{15}$$

For any architecture $a \in A$, a is determined by the vector of "edge-make" m_a, so loss function in Equation 16 can be written as $\mathcal{L}(m_a, w_a)$. As the architecture a is not deterministic but stochastic under probability distribution θ, the NAS problem is reorganized to minimize the expected value of loss function as

$$\min_{\theta} \min_{w_a} \mathbb{E}_{a\sim P_\theta}[\mathcal{L}(m_a, w_a)] \tag{16}$$

However this equation is differentiable during back propagation with respect to w_a but not with m_a, since a series of process that sampling the mask vector m_a according to probability of θ is discrete not continuous. Therefore it is incapable to backpropagate the gradient of θ through discrete random variable m_a. To enable the random variable to have continuity, GumbelSoftmax [26] is applied to m_k^{ij} as

$$m_k^{ij} = GumbelSoftmax(\theta_k^{ij}|\theta^{\mathbf{ij}}) = \frac{exp((\theta_k^{ij}+g_k^{ij})/\tau)}{\sum_k(exp((\theta_k^{ij}+g_k^{ij})/\tau))}, g_k^{ik} \sim Gumbel(0,1) \tag{17}$$

while g_k^{ij} is a random variable drawn from the Gumbel distribution. With adding g_k^{ij} it introduces stochastic into the process of approximating a discrete distribution, ensuring diversity in the values sampled during the model's training. The temperature coefficient τ restricts the behavior of the Gumbel Softmax. On the initial stage of training, setting τ extremely large enables m^{ij} to follows uniform distribution which keeps gradients have biased but low variance. On the other hand, at the end of the training τ is set near zero, as a result m^{ij} gradually becomes discrete categorical distribution and gradients have unbiased but high variance. The distribution introduced by GumbelSoftmax replace non-differentiable samples m_a, which was categorical variable to decide the probability, with a differentiable approximation. Accordingly, the gradient of the loss function now can be computed with respect to θ and refered to [25] such architecture search is called differentiable neural architecutre search (DNAS).

$$\nabla_{\theta,w_a}\mathbb{E}_{a\sim P_\theta}[\mathcal{L}(m_a, w_a)] = \mathbb{E}_{g\sim Gumbel(0,1)}[\frac{\partial\mathcal{L}(m_a, w_a)}{\partial m_a} \cdot \frac{\partial m_a(\theta, g)}{\partial\theta}] \tag{18}$$

Using DNAS, the cost of a candidate architecture a is

$$Cost(a) = \sum_{e_k^{ij}\in E}(m_k^{ij} \times \text{\#FLOP}(e_k^{ij}) \times \text{weight-bit}(e_k^{ij}) \times \text{act-bit}(e_k^{ij})) \tag{19}$$

where #FLOP($\cdot$) denotes the number of floating point operations of the convolution operator, weight-bit($\cdot$) denotes the bit-width of the weights and act-bit($\cdot$) denotes the one of the activation.

C. Training 4bit quantized model

The increasing complexity of models and datasets has led to longer Deep Neural Networks (DNNs) training times, hindering innovation. To address this, reduced-precision training, particularly 16-bit and 8-bit formats, has become popular for boosting performance and power efficiency in deep learning hardware. While 8-bit formats are deemed sufficient for training, achieving successful convergence with 4-bit representations poses challenges due to quantization errors, precision issues, and dynamic range limitations. [27] introduces a groundbreaking end-to-end solution using a novel 4-bit floating point format, rounding schemes, and gradient scaling techniques. The proposed methods facilitate model convergence with minimal accuracy loss across various deep learning benchmarks.

Not just using 4-bit size for quantizing weights and activation into 4 bit integer, [27] shows model convergence using 4-bits for all tensors including gradients. For gradients, they are quantizied into FP4. However, compared to baseline FP32, quatization into FP4 with common radix-2 showed severe accuracy degradation on ResNet18. This is because of the limitation of available quatization range. Even radix-2 FP4 format with [sign, exponent, mantissa] = [1,3,0] can handle dynamic range for 2^6. To cover broad gradient value, the format of FP4 is decided as radix-4 with [sign, exponent, mantissa] = [1,3,0], which can have dynamic range 2^{12}. This round assign gradient in a 4^n way. A value between $[4^{n-1}, 4^n]$ is chosen to be 4^{n-1} or 4^n divided by mid-point$((4^{n-1}+4^n)/2 = 4^n/1.6)$.

But still the radix-4 FP4 round format leaves critical gap between quantized values while backpropagate, it caused significant loss. The quantization error, introduced while altering the original value into radix-4 FP4 format, can be described in a manner of expectation value of the weight gradient dL/dw as:

$$E[dL/dw] = E[x \cdot (dL/du + q_{error})] = E[x \cdot dL/dy] + E[x \cdot q_{error}] \tag{20}$$

$E[q_{error}]$ will introduce a non-zero bias and skew weight tensors causing a large shift in output activations so called an internal covarivate shift (ICS). As a layer goes deeper, such trivial errors will accumulated and hinder the entire training. Batch normalization can be a nice resolution, but it is not a perfect key because of the strong dependence of loss/accuracy on the batch's mean μ_B and standard deviation σ_B.

QLoRA(Quantized Low-Rank Adaptation)[28] is an example of 4bit quantization fine tuning approach. QLoRA stands for 4 bit quantization and Low-Rank Adapters(LoRA). LoRA[29] is a traning method that freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture. In QLoRA, the original pre-trained weights of the model are quantized to 4-bit and kept fixed during fine-tuning. Then, a small number of trainable parameters in the form of low-rank adapters are introduced during fine-tuning. These adapters are trained to adapt the pre-trained model to the specific task it is being fine-tuned for, in 32-bit floating-point format.

When it comes to computations like forward and backward pass or inference, the 4-bit quantized weights are dequantized back to 32-bit floating-point numbers. Only the 32-bit LoRA weight tensor has been updated and stored in memory. This kind of fine-tuning allows the neural network structure to learn it is fine-tuned on a specific task given its actual dequantization error of every single weight tensor of 4-bit precision. After the fine-tuning process, the model consists of the original weights in 4-bit form, and the low-rank adapters in their higher precision format. With low-rank adapters retaining higher precision which are responsible for recognizing that a specific task is being held, model is able to capture more subtle patterns in the data.

QLoRA show that 4-bit quantization still have consistency with respect to ordinary 1-bit full finetuning and 16-bit LoRA finetuning performance on academic benchmarks. Nevertheless still compared with full 16-bit finetuning performance at extremely large parameter scaled for example 33B and 65B, it couldn't match comparable result. This left [28] a future work that how to find a point where performace-precision trade-off exactly lies for QLoRA tuning.

D. Ultra Quantization

Developing CNN with more deep structure enabled to achieve better performance. As noted before, the VGG-16 [21] contains lots of parameters and CNNs have even more. It means that the deep CNNs heavily depend on the high-performance hardware such as GPU. [30] On the other hand, porting those recent developments into device, such as smart-phones, tablets, TVs etc, is challenging due to limited computational resources. Embedded devices based on FPGAs, semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects, have only a few thousands of computing units, far from dealing with millions of floating-point operations in the common deep learning models. To resolve such problem, binarization introduces method using only 1 bit to represent the value throughout the neural network process using dramatically less than 4 bits. The binarized network enables memory requirement reduced by 32x compared to FP32, resulting in up to 38× speedups and 89× energy efficiency over existing frameworks on mobile GPUs [31]. Concise research have been conducted while focusing on not only minimized capacity of binarization but also degradation of accuracy.

BinaryConnect [32] opens the idea of binarization by representing weights during forward and backward propagation into +1 or -1 (1 bit). Before staring the forward propagation, the previous weights are converted into binary in a deterministic form or stochastic form. On deterministic binarization, the binarized value becomes +1 if weight value is positive and else -1. In the contrary, on stochastic binarization, the binarized value is decided according to probability distribution:

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w) \\ -1 & \text{with probability } 1 - p \end{cases}$$

and hard sigmoid, which is refered as $\sigma(w)$, determines the probability.

$$\sigma(x) = \text{clip}\left(\frac{x+1}{2}, 0, 1\right) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right) \tag{21}$$

After binarization,forward propagation is then proceeded as illustrated in Algorithm 5, where each result of activation is still real value. Because every weight is treated as +1 or -1, multiplication procedure does not exist anymore during the forward propagation. As a result of consecutive calculation by chain rule during backpropagation, gradients have real value representation which would eventually update weight into real value. To keep weight into binary, those updated weights are clipped within the [-1, 1] interval. Without this clip, large real value won't be able to exercise its impact on later binarization, so such clipping is needed working as a regularization.

Binarized NN (BNN) [33] goes one step further the BinaryConnect, applying binarization also in activation during forward propagation. If sign function, which was used to binarize weight on BinaryConnect method, is kept used when binarizing activation, its derivative function would cause significant degradation during back propagation. Derivative function of sign function, which is called delta function, stays zero except at the point 0. However to obtain down stream gradient during back propagation, gradient of sign function would be kept accumulated and if that accumulated value stays 0, the entire training will fail causing saturation effect or gradient vanishing. To resolve this, Straight-through Estimator (STE) method is proposed

Algorithm 5 SGD training with BinaryConnect. C is the cost function for minibatch and the functions binarize(w) and clip(w) specify how to binarize and clip weights. L is the number of layers. Obtained from [32].

Require: a minibatch of (inputs, targets), previous parameters w_{t-1} (weights) and b_{t-1} (biases), and learning rate η.
Ensure: update parameters w_t and b_t.

1. Forward Propagation
$w_b \leftarrow$ binarize(w_{t-1})
For $k = 1$ to L, compute a_k knowing a_{k-1}, w_b and b_{t-1}
2. Backward Propagation
Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$
For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and w_b
3. Parameter update
Compute $\frac{\partial C}{\partial w_b}$ and $\frac{\partial C}{\partial b_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}
$w_t \leftarrow$ clip(w_{t-1} - $\eta \frac{\partial C}{\partial w_b}$)
$b_t \leftarrow b_{t-1}$ - $\eta \frac{\partial C}{\partial b_{t-1}}$

which is to find approximate optimal function under certain threshold using estimator. In an empirical manner, $hardtanh$ function:

$$Htanh(x) = \text{clip}\,(x, -1, 1) = \max\,(-1, \min\,(1, x)) \tag{22}$$

was chosen to be a the estimator of sign function's derivative, whose gradient stays 1 between [-1, 1] and else 0. So the quantization is held by sign function as usual and derivative of sign function during back propagation is replaced by $hardtanh$ function.

Though those two binarization approaches brought memory efficiency, it has been pointed out that they were implemented just for heavy redundant models, AlexNet and VGG, not for already efficient models like MobileNet. [34] propose a quantization scheme that allows inference to be operated only by integer and provide a training framework minimizing the loss of accuracy from quantization especially on convolution model. On quantization scheme, real value is converted into 8-bit integer as each max and min real value is assigned to 0 and 255 respectively. The main goal of the quantization scheme is to conduct every arithmetic during inference only with integer. An uniform quantization of scale S zero-valued in Z is used. The normal convolution $a^{(i,k)} = \sum_{j=1}^{N} x^{(i,j)} w^{(j,k)} + b^{(i,k)}$ is therefore represented as:

$$a_q^{(i,k)} = Z_a + \frac{S_x S_w}{S_a} \sum_{j=1}^{N} (x_q^{(i,k)} - Z_x)(w_q^{(j,k)} - Z_w) + \frac{S_b}{S_a}(b_q^{(i,k)} - Z_b) \tag{23}$$

where S_x, S_w, S_a denote the scale factor for input, weights and activations and Z_x, Z_w, Z_a denote the zero values for input, weights and activations, respectively.

Although it tries to use integer arithmetic only during inference, operation between integer variables inevitably produce real value arithmetic. Nevertheless, to minimize the intervention of real value arithmetic from other operation, some tricks are applied. First, the combination of scaling factors is defined as multiplier M. The multiplier M needs to be coordinated as it is the only non-integer value from above equation.

$$M := \frac{S_x S_w}{S_a} \tag{24}$$

The non-integer multiplier M into integer level which can be written as:

$$M_0 = 2^n M \tag{25}$$

Computing multiplier M is not that computational burden as each scaling factors are fixed after training and so multiplier M can be computed offline. The reason to scale up the M up to 32-bit is because multiplication and accumulation between input $x_q^{(i,k)}$ and $w_q^{(j,k)}$ would occupy space up to 32-bit. Then the result between M_0 and summation is multiplied with scale down value 2^{-n}, finally to get a int 8-bit. The corresponding equation is in Equation (26). Second, related to the bias, bias vector is quantized into int32 and it uses 0 for zero point Z_{bias}. Also its quantization scale S_{bias} gets same value as $S_1 S_2$. These process seems to be conducted for simplicity during operation while abandoning slight accuracy. Summing up two tricks, Equation (23) can be written as

$$a_q^{(i,k)} = Z_a + 2^{-n}\{(2^n M)(b_q^{(i,k)} + \sum_{j=1}^{N} (x_q^{(i,k)} - Z_x)(w_q^{(j,k)} - Z_w))\} \tag{26}$$

At training, while quantization is used on forward propagation, updates are held by real value as backpropagation uses only gradients of real value.

On the experiment from [35], integer-only quantization scheme is applied on Transformer model like RoBERTa-Base and RoBERTa-Large. Compared to a baseline, it showed 0.3/0.5 average GLUE (General Language Understanding Evaluation tasks) points improvement. Also for end-to-end inference latency, there is x4 speedup compared to full precision. Therefore, it means that, on specialized hardware with efficient integer computation support, we may deploy quantized models with significantly better performance in terms of accuracy, speed and energy-consumption than using full precision version of the models. This result is too promising though. Usually, quantized models do not achieve greater accuracy than full precision ones. However, this definitely shows the potential of usage of quantization in Neural Networks, and its application in different devices.

VI. Conclusion

In this paper, we have introduced the idea of quantization and arranged specific field where quantization can be applied. Using scaling factor, dequantization is also convertibly involved. Quantization scale is largely separated into uniform quantization and non-uniform quantization according to the variance of quantized range.

In Gaussian Processes where mathematical framework is used for modeling and predicting probabilistic relationships between variables, reduced precision can be applied by using conjugate gradient re-scaling and re-orthogonalization. By quantizing kernel or using Random Fourier Features, it is possible to handle high dimensional computation easily when working with kernel.

In Neural Network model, the quantization method has been extensively studied in the state-of-art. Not only the quantization schemes for different layers has been studied (layerwise/groupwise quantization), but also quantization has been also studied for training a model taking it into account. When it comes to exponentially increasing model and parameter size on machine learning, reduced or mixed precision is a decent breakthrough. With a mixed precision, it became possible to cut down memory by storing weights into half precision while still executing arithmetic with full precision for accuracy. Down sizing precision into 4 bit allows significant inference latency improvement and extreme binary or integer quantization suggested a possibility to load heavy machine learning model into limited capacity of devices.

Based on the information provided in this review, several future research lines may be outlined. In Gaussian Processes and Kernel approximation, the state of the art is still in a premature stage, with only few research papers (to our knowledge) which have studied quantization in this techniques. For example, for Subsection IV-A a general technique of quantization was used, but no theoretical guarantee was explored and for Section III the quantization level achieved is still half precision, so studying similar use cases with lower precision could be an interesting approach. For Section V, despite being the most studied models for quantization, there are still some open questions that could be addresses in the future. For example, could there be any theoretical guarantee for a family of neural networks? Can a general guideline for quantization in Machine Learning be achieved in that case? All these questions may be discussed in future work, allowing us to further exploit the benefits of quantization in memory-saving and lowering inference time in Machine Learning.

REFERENCES

[1] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," *arXiv preprint arXiv:2007.05558*, 2020.

[2] Z. Lv, D. Chen, R. Lou, and Q. Wang, "Intelligent edge computing based on machine learning for smart city," *Future Generation Computer Systems*, vol. 115, pp. 90–99, 2021.

[3] X. Liu, J. Yu, J. Wang, and Y. Gao, "Resource allocation with edge computing in iot networks via machine learning," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3415–3426, 2020.

[4] S. Wang, M. Chen, X. Liu, C. Yin, S. Cui, and H. V. Poor, "A machine learning approach for task and resource allocation in mobile-edge computing-based networks," *IEEE Internet of Things Journal*, vol. 8, no. 3, pp. 1358–1372, 2020.

[5] M. Molina, J. Mendez, D. P. Morales, E. Castillo, M. L. Vallejo, and M. Pegalajar, "Power-efficient implementation of ternary neural networks in edge devices," *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 20 111–20 121, 2022.

[6] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," in *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 2547–2554.

[7] M. Augasta and T. Kathirvalavakumar, "Pruning algorithms of neural networks—a comparative study," *Open Computer Science*, vol. 3, no. 3, pp. 105–115, 2013.

[8] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian, "Variational convolutional neural network pruning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2780–2789.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[11] M. Croci and M. B. Giles, "Effects of round-to-nearest and stochastic rounding in the numerical solution of the heat equation in low precision," *IMA Journal of Numerical Analysis*, vol. 43, no. 3, pp. 1358–1390, 2023.

[12] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[13] C. E. Rasmussen, C. K. Williams *et al.*, *Gaussian processes for machine learning*. Springer, 2006, vol. 1.

[14] W. J. Maddox, A. Potapcynski, and A. G. Wilson, "Low-precision arithmetic for fast gaussian processes," in *Uncertainty in Artificial Intelligence*. PMLR, 2022, pp. 1306–1316.

[15] T. Zhang, "Learning bounds for kernel regression using effective data dimensionality," *Neural computation*, vol. 17, no. 9, pp. 2077–2098, 2005.

[16] D. Qin, X. Chen, M. Guillaumin, and L. V. Gool, "Quantized kernel learning for feature matching," *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[17] J. Zhang, A. May, T. Dao, and C. Ré, "Low-precision random fourier features for memory-constrained kernel approximation," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1264–1274.

[18] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," *Advances in neural information processing systems*, vol. 20, 2007.

[19] X. Li and P. Li, "Quantization algorithms for random fourier features," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6369–6380.

[20] J. Blackard, "Covertype," UCI Machine Learning Repository, 1998, DOI: https://doi.org/10.24432/C50K5N.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[22] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[23] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.

[25] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of convnets via differentiable neural architecture search," *arXiv preprint arXiv:1812.00090*, 2018.

[26] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *arXiv preprint arXiv:1611.01144*, 2016.

[27] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1796–1807, 2020.

[28] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *arXiv preprint arXiv:2305.14314*, 2023.

[29] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[30] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, 2020.

[31] G. Chen, S. He, H. Meng, and K. Huang, "Phonebit: Efficient gpu-accelerated binary neural network inference engine for mobile phones," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 786–791.

[32] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in neural information processing systems*, vol. 28, 2015.

[33] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[34] 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 IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[35] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," in *International conference on machine learning*. PMLR, 2021, pp. 5506–5518.