Fully Connected Networks on a Diet With the Mediterranean Matrix Multiplication

Hassan Eshkiki, Benjamin Mora, and Xianghua Xie, *Senior Member, IEEE*

***Abstract*— This article proposes the Mediterranean matrix multiplication, a new, simple and practical randomized algorithm that samples angles between the rows and columns of two matrices with sizes m, n, and p to approximate matrix multiplication in $O(k(mn+np+mp))$ steps, where k is a constant only related to the precision desired. The number of instructions carried out is mainly bounded by bitwise operators, amenable to a simplified processing architecture and compressed matrix weights. Results show that the method is superior in size and number of operations to the standard approximation with signed matrices. Equally important, this article demonstrates a first application to machine learning inference by showing that weights of fully connected layers can be compressed between 30× and 100× with little to no loss in inference accuracy. The requirements for pure floating-point operations are also down as our algorithm relies mainly on simpler bitwise operators.**

***Index Terms*— Matrix multiplication, neural networks, randomized algorithms.**

I. Introduction

MATRIX multiplication is at the heart of various crucial algorithms and is used by a large variety of applications. It supports many applications and scientific fields such as physics (e.g., lattice QCD), machine learning, and data science in general, where calculating correlation between variables can be important; and also, information retrieval algorithms indirectly used by many people through search engines. One aspect with matrix multiplication is that the basic algorithm's complexity does not scale linearly, which becomes problematic when processing large datasets, hence requiring expensive computational resources. Indeed, while square matrices require $O(n^2)$ storage space, $O(n^3)$ computational steps are executed by the basic algorithm.

Since the pioneering work by Strassen [1] demonstrating a subcubic complexity, many deterministic and nondeterministic techniques for matrix multiplication have been proposed, as any progress made on such a fundamental concept can have a large impact. Nevertheless, there are issues plaguing the more theoretically optimal algorithms, especially the deterministic ones. Typically, the presence of very large complexity constants coupled with algorithms, which do not benefit much from the stream-oriented modern processor architectures (versus random access), makes these low-complexity algorithms difficult to use in practical applications. In the last decade or so, randomized iterative algorithms that are able to get closer to an optimal complexity of $O(n^2)$ have gained attention. These algorithms also exhibit a run-time complexity constant as a nonlinear function of the desired precision and usually in the order of ϵ^{-2}. Nevertheless, there is a point to using approximation algorithms as some applications that handle large datasets do not necessarily need an exact solution. For instance, finding approximate correlations quickly may be preferred to finding exact solutions in firm real-time systems (e.g., financial services and high-frequency trading), especially if the error variance can be easily quantified or at least estimated. Drineas and Mahoney [2] have recently shown that various scientific areas can benefit from randomized linear algebra algorithms.

In the following, we therefore introduce a new, simple iterative randomized algorithm called the Mediterranean matrix multiplication (M^3) and show a first application to the compression of fully connected (FC) layers. In particular, our M^3 runs in $O(k(mn + np + mp))$ steps for approximating any matrix multiplication, where k is the number of trials required to reach a given accuracy. In particular, it is shown that this error decreases at a rate of $O(k^{-0.5})$. The first part of this article demonstrates some important theoretical properties of the algorithm, including reducing the approximation for square matrix multiplications to either $O(n^{2+\epsilon})$ [3] or $n^2 + O(n^2)$ steps [4] using fast rectangular matrix multiplication algorithms. This article then demonstrates that one can advantageously use the M^3 in the FC layers of a neural network when inferring from data.

To facilitate the understanding of the following, we enumerate some of the symbols used hereafter.

1) $A, B, C, Y, W,$ and X: Matrices and vectors (X and Y) such that $C \simeq AB$ and $Y = WX$ (W is a weight matrix, X is a layer input, and Y is its output).
2) A_i and W_i: Rows i of respective matrices A and W.
3) B_j: Column j of matrix B.
4) k: Number of trials/hyperplanes used [and the main tradeoff factor between precision and complexity, with the error decreasing at a rate of $O(k^{-0.5})$].
5) m: Number of rows of A, C, W, and Y.
6) p: Number of columns of B and C.
7) n: Number of columns of A and W, and number of rows of B and X. When discussing square matrices, we will assume that all dimensions are of size n.

Manuscript received 24 July 2020; revised 1 May 2021; accepted 6 May 2022. This work was supported in part by the UK Government through the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/R51312X/1 and in part by the Natural Environment Research Council (NERC) under Grant NE/W502911/1. *(Corresponding author: Benjamin Mora.)*

The authors are with the Department of Computer Science (Computational Foundry), Swansea University, Swansea SA1 8EN, U.K. (e-mail: benjamin_mora@hotmail.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2022.3176197.

Digital Object Identifier 10.1109/TNNLS.2022.3176197

8) ϵ: Error made on the approximation.
9) ϵ': A constant arbitrarily close to zero used for the complexity notation.
10) $1-\gamma$: Confidence level for the error being made.
11) ω: A complexity constant for square matrix multiplication algorithms such that their complexity is expressed as $O(n^{\omega})$.
12) α: A complexity constant for rectangular matrix multiplication such that the product of an $n \times n$ matrix by an $n \times n^{\alpha}$ matrix is known to be achievable in $O(n^{2+\epsilon'})$ [5].

II. Related Work

A. Advances in Matrix Multiplication

Deterministic algorithms for "exact" multiplication of two matrices with a reduced $O(n^{\omega})$ complexity bound have been intensively investigated in the past. Since the original work by Strassen [1] ($\omega < 2.808$), many improvements to this upper bound have been made. One can cite [6] ($\omega < 2.796$)), [7] ($\omega < 2.78$), [8] ($\omega < 2.53$), [9] ($\omega < 2.52$), [10] ($\omega < 2.5$), [11] ($\omega < 2.4785$), [12] ($\omega < 2.375$), [13] ($\omega < 2.374$), [14] ($\omega < 2.373$), and [15] ($\omega < 2.3728639$). Interesting results have also been obtained in the area of rectangular matrix multiplication. For instance, the $\mathbb{R}^{n\times n} \times \mathbb{R}^{n\times \log n}$ product of two matrices can be implemented in $n^2 + O(n^2)$ steps [4]. It has then been demonstrated that multiplying an $n \times n$ matrix by an $n \times n^{\alpha}$ matrix can be completed in $O(n^{2+\epsilon})$ arithmetic operations [3] for any $\epsilon > 0$ (and any sufficiently large matrices with $n > N_{\epsilon}$) if α is less than 0.172. Later, the value of α was raised to approximately 0.29462 [5]. Recently, Le Gall [16] demonstrated that multiplying an $n \times n^{\alpha}$ matrix by an $n^{\alpha} \times n$ matrix could exhibit an $O(n^{2+\epsilon})$ complexity with α being less than 0.30298. Finally, there is an important duality theorem for matrix multiplications stating that the number of multiplications involved for computing matrix products involving three fixed dimensions m, n, and p in any order is constant [17]. As such, computing an $\mathbb{R}^{n\times n} \times \mathbb{R}^{n\times f(n)}$ product requires as many multiplications as computing an $\mathbb{R}^{n\times f(n)} \times \mathbb{R}^{f(n)\times n}$ product. These results on rectangular matrices are essential to demonstrate an improvement on the current bounds of approximated matrix multiplication algorithms.

While not improving theoretical bounds, research restricted to Boolean matrix multiplication has nevertheless led to specific complexity results and produced more practical algorithms. The famous four-Russian algorithm [18] led to a reduced complexity of $O(n^3/\log^2 n)$ for multiplying Boolean matrices. This result was superseded by a complexity of $O(n^3(\log\log n)^2/\log^{9/4} n)$ in [19]), $O(n^3(\log\log n)^3/\log^3 n)$ in [20], and finally $\hat{O}(n^3/\log^4 n)$ in [21]—currently the best known bound of this type.[1]

From a purely theoretical point of view, other ways to reduce the strict number of arithmetic operations used for performing matrix multiplication exist. $O(n^2 \log n)$ was demonstrated in [22], while a better bound of $O(n^2)$ was achieved in [23] and [24]. However, these results are achieved by using extra-large integers or real values that usually have a binary size in the order of $O(n)$. This allows "packing" several operations into a single theoretical arithmetic operation that cannot, however, be executed in $O(1)$ steps on a regular computer. In general, while the lowest bound value for ω is not yet known, many have conjectured that the true value for ω is 2 for a deterministic algorithm.

Nondeterministic algorithms have also been studied extensively. If one wants to compute an actual product of two matrices to a given precision, there are several iterative solutions exhibiting an $O(n^2)$ complexity per iteration for calculating the estimation. This obviously implies a tradeoff between the complexity constant and the final precision wanted, with the precision factor until now being included in the complexity result as a representation for the extra number of iterations needed. While not all applications using matrix multiplications are able to deal with some error margins, there are important areas where such an error could be acceptable. For instance, one can approach the row/column dot product value defined as $\sum_{k=1}^{n} a_{ik}b_{kj}$ by considering a random subset of all the products $a_{ik}b_{kj}$ as an approximation, with increasing accuracy as $n \to \infty$. This selective sampling approach is sensitive (i.e., exhibiting unbounded variance), however, to the input values of the two matrices [25], [26], usually when high frequencies are present in the data. To circumvent this problem, it was proposed in [27] and [28] to select a limited number of dimensions using an importance sampling scheme in accordance with the lengths of rows of A and columns of B, which guarantees a bounded variance for the estimation.

A different, more streaming-oriented approach is the one proposed in [26] and based on random projections principles as described by Johnson and Lindenstrauss [29]. At the heart of this technique is the computation of an $ASS^{\top}B$ product, where S is a Johnson–Lindenstrauss transform (JLT) sign matrix. Indeed, Sarlos [26] demonstrated that to achieve an error less than ϵ with a confidence level of $1-\gamma$ (i.e., ensuring $\Pr(||AB-C||_F < \epsilon ||A||_F ||B||_F) \geq 1-\gamma$), the number of steps required is of the order of $O((mn+np+mp) \times (\log(1/\gamma) \times \epsilon^{-2} + \log(1/\gamma)^2))$. This bound was further improved [30] to $O((mn+np+mp) \times \log(1/\gamma) \times \epsilon^{-2})$, which can theoretically be reduced to O(n^2) steps for sufficiently large n values if fast rectangular matrix multiplications are used.

B. Compression of Neural Networks

Matrix multiplication is the most time-consuming operation in neural networks, especially when using FC layers, and therefore, it is not surprising that extensive work has been done to optimize this part of the learning process. One popular way to do so is to use lower precision calculations, with register sizes between 4 and 16 bits being implemented nowadays on specialized hardware. A popular format showing little loss in overall precision is bfloat16, which is currently preferred to the more standard IEEE 16-bit half-precision floating-point format in machine learning applications [31].

While it is unclear what the best precision to use is and what are the reasons behind this, some researchers have been able to push it to a limit of 1 bit by using binary network models. In Binary-Connect, Courbariaux *et al.* [32] provided a training algorithm to restrict weights to { 1,-1 },

[1]The $\hat{O}$ notation is used to remove the poly$(\log\log n)$ factor in the expression of the complexity.

therefore improving storage by a factor of 32 (compared to FP32) with little impact on the error rate. As integer or Floating Point (FP) precision calculations were still required in some calculations, XNOR-Nets were proposed in [33], allowing binarized (0 or 1) weights, operations, and input. Recently, a more flexible model using a variable number of bits (e.g., 1–3) has been proposed in [34].

As reducing precision reaches its limits quickly, other approaches have focused on compressing matrices as a whole. A survey of some of the approaches is provided in [35]. A typical way to compress matrices in scientific applications is to use a low-rank approximation, where the less important eigenvectors of the kernel are typically removed [36]–[38]. Novikov *et al.* [39] improved on low-rank approximation by proposing tensor decomposition and demonstrated several-fold improvements in the compression of FC layers.

Random projection methods in machine learning have been popularized with kitchen sinks [40], making kernel methods more scalable. The fastfood transform [41] and then deepfried networks [42] improved on kitchen sinks by reducing memory space and processing time. At the heart of these methods is a diagonal random matrix [thus reducing storage costs from $O(nd)$ to $O(n)$] combined with fast transforms that can approximate the weight matrix of an FC network [42].

Overall, the Mediterranean diet we are introducing in Section IV has similar advantages to some of these approaches [33], [40]–[42] for inference as it combines predominant binary operators, low-rank matrices, and random projections in one framework.

III. Monte Carlo Sampling of Angles Between Vectors

This section introduces a complexity analysis for the M^3 algorithms proposed in this article. At the heart of these algorithms is a relatively simple principle of randomly sampling the angle between two vectors. This principle was initially introduced in [43] to provide a good approximation to the solution of the max-cut problem, but curiously never found its way into a matrix multiplication algorithm. In this article, it was noticed that the probability of having a random plane separating two vectors (i.e., the probability of getting opposite dot-product signs) was proportional to the angle formed by these two vectors. Others have since used this principle successfully and applied it to produce algorithms identifying similarities (e.g., [44] with the aim of providing good hash functions). This work finds a new scope of application for this idea by subsequently integrating it inside a simple Monte Carlo process, eventually leading to a lower matrix multiplication complexity bound.

The starting point of our Monte Carlo evaluation is the expression of a dot product between two vectors as

$$C_{i,j} = A_i B_j = ||A_i|| \cdot ||B_j|| \cdot \cos\theta_{ij}. \tag{1}$$

From there, one notices that the angle value θ_{ij} is the only limiting factor in establishing a minimal complexity result for matrix multiplication. Indeed, all the necessary $||A_i|| \cdot ||B_j||$ vector norm products can be calculated in $O(n^2)$ steps. If we

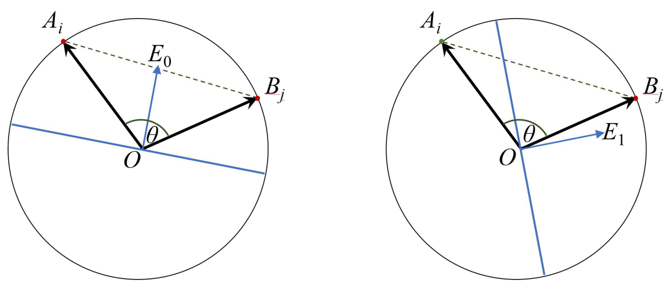


Fig. 1. Illustration of Monte Carlo sampling of an angle between two angles, which accumulates results from random tests (Algorithm 1). Left: both points are on the same side of the hyperplane and test will return 0. Right: hyperplane is separating the two points and the test will return +1.

can estimate the angle for every entry $C_{i,j}$ in k steps with a simple Monte Carlo sampling process, then we know that the variance in the estimation will be proportional to $1/k$. Furthermore, if we can compute a given number of k trials "for free" (i.e., in constant time per entry) as n increases, with a relationship $k = f(n)$ and f being a monotonic function that tends to infinity, the relative error made can then be factored out of the complexity expression. This allows the existence of a lower complexity bound for matrix multiplication approximation, with our main theorem expressed as follows.

Theorem 1: Let ϵ, ϵ', and γ be three positive real values arbitrarily close to 0. The product of two square matrices A and B can be approximated as a matrix C with an algorithmic complexity equivalent to that of a rectangular matrix multiplication (i.e., either $O(n^{2+\epsilon'})$ or $n^2 + O(n^2)$ steps) while satisfying $\Pr(||C - AB||_F < \epsilon||A||_F||B||_F) \geq 1 - \gamma$.

While a similar result has already been given in [30], we will demonstrate that it is still valid when sampling angles instead of using randomly signed matrices.

A. Sampling Principles

Lemma 2: Let A_i and B_j be two noncollinear vectors of any length and $\mathcal{P}$ be the 2-D plane defined from a linear combination of these two vectors and the origin, as shown in Fig. 1. The angle between these two vectors can be approximated with a Monte Carlo simulation (Algorithm 1) that initializes the angle to 0 and then repeats k times: 1) choosing a random line $\mathcal{L}_\phi$ of $\mathcal{P}$ crossing the origin and 2) adding π/k to the angle when $\mathcal{L}_\phi$ separates the two vertices A_i and B_j.

The reason for this lemma to hold is straightforward and many aspects are already developed in [43]. We will detail the proof and analyze some properties further for inclusiveness.

Proof: Let θ be the angle between two unit vectors as defined by the arc cosine of the vectors' dot product and within the range $[0..\pi]$ (i.e., the complement of the reflex angle). This angle can be defined as shown in Fig. 1 by the integral over the section related to the angle

$$\theta = \frac{1}{2}\int_0^{2\pi} \operatorname{Box}(A_i, B_j, \mathcal{L}_\phi)d\phi \tag{2}$$

where Box is a box function that returns 1 if the line $\mathcal{L}_\phi$ separates the two vertices A_i and B_j and 0 otherwise. Note

Algorithm 1 Iterative Dot Product Approximation by Angular Sampling

```
Input: Row $A_i$, column $B_j$ and a number of iterations $k$.
Output: An approximation $C_{ij}$ such as $C_{ij} \simeq A_i B_j$.

$C'_{ij} = 0$
$NormA_i = ||A_i||$
$NormB_j = ||B_j||$
for $s = 1$ to $k$ do
    $E_s = RandomNormalDist_n(\sigma = cst, \mu = 0)$
    $DotA_i = A_i \cdot E_s$
    $DotB_j = E_s^T \cdot B_j$
    if $(DotA_i \cdot DotB_j < 0)$ then
        $C'_{i,j} = C'_{i,j} + 1$
    end if
end for
$C_{i,j} = \cos(\frac{\pi}{k} C'_{i,j}) \cdot NormA_i \cdot NormB_j$
```

that since $\mathcal{L}_\phi$ is equivalent to $\mathcal{L}_{\phi+\pi}$, the final result needs to be divided by a factor of 2. The mathematical principles to estimate this integral using a Monte Carlo simulation are well established and θ can be evaluated as

$$\hat{\theta} = \sum_{t=1}^{k} \frac{\pi}{k} \text{Box}\big(A_i, B_j, \mathcal{L}_{\phi_t}\big)\big) = \frac{\pi}{k} \sum_{t=1}^{k} \text{Box}\big(A_i, B_j, \mathcal{L}_{\phi_t}\big)\big) \tag{3}$$

where k is the number of random tests desired, with $k > 0$. Finally, we have been assuming that the vectors are not collinear as we would not be able to define a 2-D plane otherwise. If this is the case, A_i and B_j will always be either on the same or opposite side of the separating line, and the approximated value for the angle will respectively be either 0 or π, which is as expected. □

It is useful to determine the statistical properties of such a test (Algorithm 1), which can be established from classical statistical analysis. If we assume that the $\mathcal{L}_\phi$ space is sampled uniformly, then our box test is a Bernoulli trial of well-known success probability θ/π and performing k samplings will result in a binomial distribution, which leads to the following lemma.

Lemma 3: There exist a sufficient number of iterations k, an error margin ϵ decreasing as k increases, and a confidence level $1 - \gamma$ such that

$$\Pr\big(|C_{i,j} - A_i B_j| \leq \epsilon \cdot ||A_i|| \cdot ||B_j||\big) \geq 1 - \gamma. \tag{4}$$

Proof: The main statistical property of interest to us is the evaluation of the error made for the estimation within some level of confidence. First, as we are dealing with a well-established sum of Bernoulli trials (i.e., a binomial distribution), the variance on the estimation $\hat{\theta}$ is given by

$$\begin{aligned}\text{Var}\big[\hat{\theta}\big] &= \text{Var}\left[\sum_{t=1}^{k} \frac{\pi}{k} Box\big(A_i, B_j, \mathcal{L}_{\phi_t}\big)\right] \\ &= k\frac{\pi^2}{k^2}\text{Var}\big[\text{Box}\big(A_i, B_j, \mathcal{L}_{\phi_t}\big)\big] = \frac{\pi^2}{k}\frac{\theta}{\pi}\left(1 - \frac{\theta}{\pi}\right).\end{aligned} \tag{5}$$

Also, the expected value $\mathbf{E}(\hat{\theta})$ is trivially equal to θ. Note that

$$\frac{\theta}{\pi}\left(1 - \frac{\theta}{\pi}\right) \leq \frac{1}{4},\ \theta \in [0..\pi]. \tag{6}$$

Therefore, the maximum variance is achieved for $\theta = \pi/2$ and is bounded by

$$\text{Var}\big[\hat{\theta}\big] \leq \frac{\pi^2}{4k}. \tag{7}$$

Hence, the error on the real angle is expected to decrease proportionally to the square root of the number of trials k. Note that we have voluntarily removed the $\theta/\pi\ (1 - \frac{\theta}{\pi})$ factor when generalizing this result to all possible angles later on. Should we have some *a priori* knowledge about the minimal or maximal values of the row–column angles, such a constant could be reintroduced in the subsequent estimation of the error bound.

A similar $1/k$ convergence rate is encountered in algorithms exposed in [28] and [30], with the exception that it is now weighted by a constant in the range $[0..\pi^2/4]$. Small angles will therefore require fewer iterations, while angles close to $\pi/2$ will need $2.46\times$ more samples to reach the same error level.

We are now interested in the statistical error ϵ that results from approximating an angle with k trials, in conjunction with a confidence level of at least $1 - \gamma$. This can be expressed as

$$\Pr\big(|\hat{\theta} - \theta| \leq \epsilon\big) \geq 1 - \gamma. \tag{8}$$

From the variance and expected value of the Bernoulli trials, one can use Chebyshev's inequality to obtain a bound for the error defined as

$$\Pr\big(|\hat{\theta} - \theta| \leq \epsilon\big) \geq 1 - \frac{\theta(\pi - \theta)}{k\epsilon^2}. \tag{9}$$

The ϵ value is therefore linked to the chosen number of iterations k and the confidence level γ as follows:

$$\epsilon_{k,\gamma} = \sqrt{\frac{\theta(\pi - \theta)}{k\gamma}},\ \gamma_{\epsilon,k} = \frac{\theta(\pi - \theta)}{k\epsilon^2} \text{ and } k_{\epsilon,\gamma} = \frac{\theta(\pi - \theta)}{\gamma\epsilon^2}. \tag{10}$$

The actual property we are interested in is the estimation of the error made on $\cos\hat{\theta}$ as it is the value needed to compute the final dot product evaluation. The cos function is monotonic in the range $[0..\pi]$, with $|\cos'\theta| \leq 1$. Therefore,

$$\forall\theta_1, \theta_2 \in [0..\pi] : |\cos\theta_1 - \cos\theta_2| \leq |\theta_1 - \theta_2| \tag{11}$$

which simply implies

$$\Pr\big(|\cos\hat{\theta} - \cos\theta| \leq \epsilon\big) \geq 1 - \frac{\theta(\pi - \theta)}{k\epsilon^2} \geq 1 - \gamma. \tag{12}$$

We conclude that for any ϵ and γ in the range $]0..1[$, there exist a number of iterations $k_{\epsilon,\gamma}$ such that one can estimate the value $\cos\hat{\theta}$ with an error margin ϵ at a level of confidence $1 - \gamma$. For any given confidence level, the decrease rate in the error is proportional to $1/\sqrt{k}$. It follows that the error made on the evaluation of each $A_i \cdot B_j$ product is bounded with a given confidence level expressed as

$$\begin{aligned}\Pr\big(|C_{i,j} - A_i B_j| \leq \epsilon \cdot ||A_i|| \cdot ||B_j||\big) &\geq 1 - \frac{\theta(\pi - \theta)}{k\epsilon^2} \\ &\geq 1 - \gamma.\end{aligned} \tag{13}$$

□

Supposing that there is an algorithm allowing this bound for every entry of the result matrix, the global error bound needs to be expressed in relationship with the Frobenius norms of A and B to be comparable with other results in the area.

Lemma 4: There exists a sufficient number of iterations k that verifies

$$\Pr(||AB - C||_F \leq \epsilon ||A||_F ||B||_F) \geq 1 - \gamma. \tag{14}$$

Proof: We have

$$\begin{aligned}&\Pr(||AB - C||_F \leq x)\\ &\geq \Pr\left(\sum_{i=1,j=1}^{m,p} \left(||A_i|| \cdot ||B_j|| \cdot |\cos(\hat{\theta}_{ij}) - \cos(\theta_{ij})|\right)^2 \leq x^2\right)\\ &\geq \Pr\left(\sum_{i=1,j=1}^{m,p} \left(||A_i|| \cdot ||B_j|| \cdot |\hat{\theta}_{ij} - \theta_{ij}|\right)^2 \leq x^2\right)\end{aligned} \tag{15}$$

where x is a positive value. We now calculate the expected value of the inner sum

$$\begin{aligned}&\mathbf{E}\left(\sum_{i=1,j=1}^{m,p} \left(||A_i|| \cdot ||B_j|| \cdot |\hat{\theta}_{ij} - \theta_{ij}|\right)^2\right)\\ &= \sum_{i=1,j=1}^{m,p} \left(||A_i||^2 ||B_j||^2 \cdot \mathbf{E}\left((\hat{\theta}_{ij} - \theta_{ij})^2\right)\right)\\ &= \sum_{i=1,j=1}^{m,p} \left(||A_i||^2 ||B_j||^2 \cdot \left(\mathbf{Var}(\hat{\theta}_{ij} - \theta_{ij}) + \mathbf{E}(\hat{\theta}_{ij} - \theta_{ij})^2\right)\right)\\ &= \sum_{i=1,j=1}^{m,p} \left(||A_i||^2 ||B_j||^2 \cdot \mathbf{Var}(\hat{\theta}_{ij})\right).\end{aligned} \tag{16}$$

We know from (7) that the variance on every angle estimation can be bounded, leading to

$$\begin{aligned}&\mathbf{E}\left(\sum_{i=1,j=1}^{m,p} \left(||A_i|| \cdot ||B_j|| \cdot |\hat{\theta}_{ij} - \theta_{ij}|\right)^2\right)\\ &\qquad \leq \frac{\pi^2}{4k} \sum_{i=1,j=1}^{m,p} ||A_i||^2 ||B_j||^2 \leq \frac{\pi^2}{4k} ||A||_F^2 ||B||_F^2.\end{aligned} \tag{17}$$

Markov's inequality can now be used to finalize an upper bound. Combining (15), (17), and (18), we get

$$\begin{aligned}&\Pr(||AB - C||_F \leq x)\\ &\quad \geq 1 - x^{-2}\mathbf{E}\left(\sum_{i=1,j=1}^{m,p} \left(||A_i|| \cdot ||B_j|| \cdot |\hat{\theta}_{ij} - \theta_{ij}|\right)^2\right).\end{aligned} \tag{18}$$

Let x be $\epsilon ||A||_F ||B||_F$. We finally obtain

$$\Pr(||AB - C||_F \leq \epsilon ||A||_F ||B||_F) \geq 1 - \frac{\pi^2}{4k\epsilon^2} \geq 1 - \gamma. \tag{19}$$

We therefore conclude that for any values ϵ and γ in the range $]0..1[$, there exists a large enough integer k verifying the hypothesis. □

Even though this proof requires k to be in the order of $O(\epsilon^{-2}\gamma^{-1})$ as it is derived from a Markov inequality, it is well known that binomial distributions will lead to a $\epsilon^{-2}\log(1/\gamma)$ bound when k is finite, which is similar to the best known bound given in [30].

B. Basic Algorithm

We now extend the randomized Algorithm 1 respecting the bound described in Section III-A to all entries of matrices A and B. Let vectors A_i, B_j, and E_s be three noncollinear vectors of $\mathbb{R}^n$, with E_s chosen randomly on the hypersphere centered on the origin and defining a unique orthogonal hyperplane $\mathcal{P}_s$. The intersection of $\mathcal{P}_s$ with the 2-D subspace $\mathcal{P}'_{i,j}$—defined from a linear combination of vectors A_i and B_j—provides a random line $L_{i,j,s}$ in $\mathcal{P}'_{i,j}$ that crosses the origin. It also ensures randomness with a uniform distribution over the angle, as initially demonstrated by [43]. As such, $L_{i,j,s}$ is our random, uniformly distributed line that can be used for sampling the angle, which also follows the statistical convergence properties enunciated earlier. We can now make the whole process efficient with $O(mn + np + mp)$ steps per iteration by reusing the very same $\mathcal{P}_s$ hyperplane for all dot products occurring in a matrix multiplication. Note that choosing a random plane ensures the uniform distribution of $L_{i,j,s}$ for all the $\mathcal{P}'_{i,j}$ planes. Hence, the basic test simply consists first of computing the signs of all dot products $A_i E_s$ and $E_s^T B_j$, which requires $O(mn+np)$ operations per random split. Obtaining opposite signs at a given s iteration, with $s \in [1..k]$, means that $\mathcal{P}_s$ is a plane separating vectors A_i and B_j. This sign test will need to be repeated $O(mp)$ times (or $O(n^2)$ times in the context of square matrices) to cater for all possible dot product combinations (A_i, B_j). Hence, the complexity of the basic algorithm over k iterations is $O(k(mn+np+mp))$. Finally, once all the angle cosines have been sampled, a scaling process of complexity $O(mn + np + mp)$ will multiply each result entry by the respective norms $||A_i||$ and $||B_j||$.

To generate a uniform and unbiased distribution on the hypersphere as required by the algorithm, it suffices to generate vector E_s entries randomly using a random number generator (RNG) following a normal distribution (i.e., with a fixed parameter σ and $\mu = 0$) for each dimension independently. Importantly, the likelihood of choosing E_s orthogonal to A_i or B_j becomes infinitely small as the range of distinct floating-point values tends to infinity. One limitation with this algorithm though is that the error made on each $C_{i,j}$ entry will decrease at a rate of $k^{-1/2}$, similar to results obtained in [28] and [30].

C. Reformulated Algorithm and Complexity Bounds

As mentioned by Clarkson and Woodruff [30], to improve on the trivial complexity bound obtained in (19), one needs only to notice that: 1) the error decreases as the number of iterations k increases and 2) Algorithm 1 can be broken down into three rectangular matrix multiplications, which allows us to compute k iterations for "free" (i.e., with the same algorithmic complexity of computing one iteration,) where $k << n$ but k also increases with n. To do so, we can write the whole algorithm using matricial products as in Algorithm 2.

To allow acceleration of rectangular matrix multiplications, an integer k must be chosen for instance such that $k = \text{floor}(n^{\alpha})$ (cf., algorithms from [5] and [16]) or such that $k \leq \log n$ [4]. Indeed, Algorithm 2 in its first stage multiplies the $\mathbb{R}^{n\times n}$ matrix A with an $\mathbb{R}^{n\times k}$ matrix E made of k random vectors of $\mathbb{R}^{n}$ and similarly multiplies B with $E^{\top}$. The second stage, which counts the number of separating planes, will require $2kn$ sign extraction operations to be performed beforehand. This thresholding will process every entry of the two temporary matrices AE and $E^{\top}B$ such that their respective entries are set to 1 when $(AE)_{i,j} > 0$ and $(E^{\top}B)_{i',j'} < 0$, and -1 otherwise. Finally, the two thresholded sign matrices of respective sizes $\mathbb{R}^{n\times k}$ and $\mathbb{R}^{k\times n}$ will be multiplied together, which, if implemented as a rectangular matrix product, is also doable with a complexity of either $O(n^{2+\epsilon'})$ if $k < 0.30298$ [16] or $n^2 + O(n^2)$ for $k \leq \log n$ [4]. This also completes the proof for Theorem 1.

Algorithm 2 Algorithm 1 Rewritten as a Product of Three Rectangular Matrix Multiplication Resulting in a Theoretical $O(n^{2+\epsilon'})$ or $n^2 + O(n^2)$ Algorithmic Complexity for Square Matrix Multiplication

```
Input: Matrix $A$ of rows $A_i$, Matrix $B$ of columns $B_j$.
Output: The resulting matrix entries $C_{ij}$ such as $C \simeq AB$.

for $i = 1$ to $n$ do
    $NormA_i = ||A_i||$
    $NormB_i = ||B_i||$
end for
$k = f(n)$                ▷ $k = floor(n^{\alpha})$ or $k = floor(log_2(n))$
$E = RandomNormalDist_{n,k}(\sigma = cst, \mu = 0)$   ▷ $O(n^2)$
$A' = AE$                 ▷ Rectangular Matrix multiplication
$B' = E^{\top}B$          ▷ Rectangular Matrix multiplication
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $A'_{i,j} = (A'_{i,j} > 0)\ ?\ 1 : -1$
        $B'_{i,j} = (B'_{i,j} < 0)\ ?\ 1 : -1$
    end for
end for
$C' = A'B'$               ▷ Rectangular Matrix multiplication
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $C_{i,j} = \cos(\frac{\pi(C'_{i,j}+k)}{2k}) \cdot NormA_i \cdot NormB_j$
    end for
end for
```

IV. Compressing FC Layers

A. Inferring With the M^3

Section III has defined a way to reduce the number of operations for multiplying matrices at a cost of introducing a noticeable error in the solution. A natural question is whether this algorithm can still find a place in applications bounded by tensor operations, such as deep neural networks (DNNs). While training models will be discussed later on, we also interestingly managed to compress the FC layers of a DNN satisfactorily for inferring data.

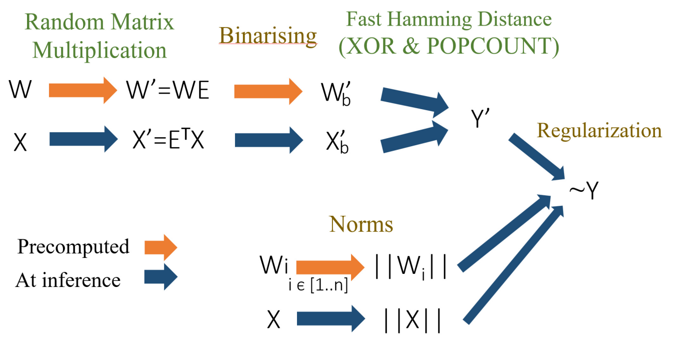


Fig. 2. Pipeline of operations for estimating the product $Y \approx WX$. Operations needed during inference are shown in blue. Note that the entries of the constant matrix E follow a normal distribution and do not require storing.

As an input X is progressing through the layers of a neural network, it follows a sequence of matrix multiplications. For an FC layer, we can write this operation as $Y = WX$, where matrix W represents the weights an FC layer and X represents the input. Replacing a standard matrix multiplication with the M^3 one requires the calculation of WE and $E^{\top}X$, where E is a random matrix of k columns, and binarizing these results similar to Algorithm 2. The pipeline for this is given in Fig. 2. We therefore need to calculate first $\text{Bin}(WE)$ and $\text{Bin}(E^{\top}X)$, with

$$\text{Bin}(X) = \{\,\text{Bin}(X_i)\,\},\ \text{with}\ \text{Bin}(X_i) = \begin{cases} 1, & X_i \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

Note that the Bin operator, unlike in Algorithm 2, is applied symmetrically to both matrices as we will later optimize code with XOR ($\oplus$) and POPCOUNT operators instead of using multiply and add operators. It is now trivial that $\text{Bin}(WE)$ can be precalculated before inferring as both matrices W and E are already known. Therefore, we can store the FC layer as a stream of binary data that, depending on the chosen number of hyperplanes k, may become significantly smaller than the original weight matrix W. We also need to store the vector of norms $\{|W_i|\}$ for the last step of the algorithm, but this space is almost negligible as this represents only a single floating-point value per row of W.

We would logically need to store a copy E as well to multiply it with the input X that is unknown at this stage. We do, however, have two options here given the random nature of this matrix. We can either create the random numbers deterministically and on the fly or store a single column vector of random numbers and rotate this vector to generate up to $n-1$ extra columns of E, in the same spirit as the FastFood transform [41] or Deep Fried networks [42]. It is important to notice that the latter gives us the opportunity to compute the product $E^{\top}X$ with a low number of floating-point operations by computing it in the frequency domain. As such, the complexity of the FC layer is mainly bounded by the operation combining the two binary streams obtained, which is represented in Algorithm 2 by a standard matrix multiplication but can be implemented from simple XOR ($\oplus$) and POPCOUNT operators. These two operators require far fewer transistors than floating-point units and can be done on 32 or

64 bits at a time on modern hardware. Hence, the M^3 does not only provide a way to compress FC layers when inferring but also provides a simplified logic with far less emphasis on floating-point operations.

B. Practical Considerations

As improvements in the complexity of matrix multiplications are sometimes not practical, this section discusses the details behind an implementation of matrix multiplication approximation algorithms on modern architectures such as GPUs. Algorithm 2 can be broken down into three stages. The first stage will compute WE and $E^\top X$, where E is a matrix made of k columns. It would therefore seem natural for k to be smaller than n as, otherwise, the computing effort would be similar to that of a basic "exact" matrix multiplication algorithm. This, however, is not a strict requirement for our algorithm as it performs most operations bitwise, and therefore, any value $k < 32n$ could be beneficial. One could also use an RNG to create E, removing the need to store this matrix. However, we can actually process up to $k = n$ separating hyperplanes by convolution in $mn \log n$ (WE) and $n \log n$ steps $(E^\top X)$ if E is chosen as a Toeplitz matrix with columns defined from a rotated random vector. Indeed, our hyperplanes must be chosen independently, and it is easy to see that the columns of a randomized Toeplitz matrix satisfy $\mathbf{E}(E_i E_j) = 0, i \neq j$. Furthermore, using a Toeplitz matrix, we only need to create and store the first column E_0 of E and possibly every subsequent column E_i such that $i \pmod{n} = 0$ (n is the number of columns of our weight matrix W).

In the next stage, thresholding of WE and $E^\top X$ can be performed in $O(2kn)$ operations and thus has no influence on the overall complexity of the process. The last stage is what differentiates our algorithm in practical terms from other algorithms such as Clarkson and Woodruff's approach [30]. As thresholding is not embedded in the random matrix but comes later in the pipeline, the final operation is a pairwise computation of Hamming distances of complexity $O(\text{kmn})$, which is simpler to carry out than a full-blown matrix multiplication. It can, for instance, be implemented with Boolean matrix multiplication algorithms such as the four-Russian algorithm [18], which would reduce the amount of computations by a $\log n$ or even a $\log^2 n$ factor for practical matrix sizes. However, this will require implementing lookup tables and adding various barriers in the flow of operations, therefore limiting parallelism. Modern processor architectures, however, include population count instructions that can output sums of 32 bits (e.g., NVIDIA CUDA cores) or 64 bits (e.g., all modern X86 CPUs) integers at every clock cycle. Being able to process so many elements at once natively is likely to be competitive with any algorithmic speedup we could get from Boolean matrix multiplications. All in all, our Mediterranean multiplication requires one XOR, one POPCOUNT, and one ADD to process 64 planes in parallel, instead of requiring a single fused multiply–add (FMA) [30] for every plane, but also requires approximately 2.46× more samples in the worst case to obtain the same variance as in [30]. While we will not analyze energy efficiency, we also hypothesize that the underlying circuits to implement $\oplus$, integer ADDs, and POPCOUNT operations are much simpler than FMA circuits and could consume less energy overall if implemented on a specialized circuit.

C. Training the Compressed Neural Network

Training needs to reflect the changes we have made to the way our FC layers work. This section describes how we ensure that a graph-based tool like Tensorflow is still able to learn patterns correctly once we are introducing our Mediterranean multiplication.

We initially train the neural network in a standard way. For some datasets (MNIST), we also augment the training dataset by adding rotated and translated samples to provide better results [45]–[47]. We then consider that convolutional layers (CLs), if present in the model, are trained and we focus on replacing the standard FC layers' multiplications with the new M^3. Indeed, FC layers are usually located at the end of a neural network. For each FC layer l, we associate a unique constant random matrix E^l with entries following a normal distribution. This matrix is in our tests generated from a deterministic RNG along with a unique seed and is created either on the fly (no storage requirement) or just from the stored columns $E^l_{i \pmod{n} = 0}$ of E^l as all other columns E^l_i can be calculated from a single-hop rotation of E^l_{i-1}. Rotating columns is preferable to just generating all the numbers from an RNG as it allows making use of the FFT algorithm to perform matrix multiplications efficiently as mentioned in Sections IV-A and IV-B.

From there, we can process forward propagation and backpropagation as follows, assuming that the CLs are already trained and the output they produce will not change for the training dataset. The forward propagation is calculated by simply replacing the regular matrix multiplication by our M^3 variant in the FC layers, hence computing WE and $E^T X$ at this stage. We, however, keep the regular matrix multiplication algorithm when performing backpropagation because the M^3-based pipeline used for compression is not differentiable, which has also the benefit of learning the extra level of error introduced in the forward pass without introducing new errors in the backpropagation step. We then simply use the weight matrix W as the gradient for backpropagation in our Tensorflow implementation.

V. Results

A. Testing Environment

All tests are performed on an Intel 4770 K processor running at 3.9 GHz and coupled with 16 GB of RAM and an Nvidia GeForce GTX 1080Ti Graphics card (11 GB). Tensorflow 2.0 is used as the artificial intelligence (AI) framework and runs on a Linux distribution.

B. Synthetic M^3 Results

1) Error Analysis: To test the M^3, we create two matrices A and B using RNGs that are part of the C++11 standard library. Entries of these two matrices match a normal distribution with $\theta = 1$ and $\mu = 0$, except for one specific test where we analyze the effect of μ on the final error obtained. While it is difficult to allow for all possible distributions arising from specific circumstances, a normal distribution was thought to be representative of various processes. It must be noted that the content of the two matrices A and B itself should not affect performance but could potentially affect the accuracy of final approximation.

All floating-point computations are performed using the standard IEEE 32-bit precision. Errors in the approximation are measured after executing a full matrix multiplication and computing $||C - AB||_F/||A||_F||B||_F$, where AB is obtained from a CUBLAS kernel call and C is the final estimation. Results are compared both to standard matrix multiplication and to $ASS^\top B$ [30], where S is a random sign matrix.

As expected from our theoretical analysis, the error decreases linearly according to the square of the number of iterations when using normally distributed inputs (Fig. 3). It can also be seen that Clarkson and Woodruff's method exhibits a lower error for the same number of iterations, with a measured error ratio close to $\pi/2$. This means that to get a similar error, one needs to compute 2.46× more planes with our algorithm, which still compares favorably as our pipeline is processing 32 or 64 bits per instruction versus one. However, these variance results are obtained for the worst case scenario where rows of A and columns of B are generated with $\mu = 0$, resulting in almost orthogonal vectors. By varying μ (Fig. 3(e) so that these rows and columns become more correlated, the error produced by our technique is actually reduced dramatically and tends to 0, a direct consequence of the Bernoulli trials [cf. (5)]. This is an important fact as it demonstrates the significant superiority of sampling angles when processing correlated data over the original method of signed matrices. One can also observe some variance in the final error made when the μ parameter becomes large. This can be explained as follows. In general, the obtained errors are very close to the theoretical ones for $\mu = 0$ as there is usually very little variation over the error measured due to the large number of entries in the final matrices. However, signed matrices perform badly in that regard when rows of A and columns of B become similar as the entries of C tend to be highly correlated due to computations becoming very similar for all entries. Our algorithm may also more subtlety inherit this problem, but as the error tends to zero, so does the error variance, which makes it more stable than the standard $ASS^\top B$ approach. It must finally be noted that we used $n = k = 8192$ in this particular test as using single-float precision provides limited accuracy, which may affect the error calculation. This comes to light in Fig. 3(d) where the error ratio between the two techniques starts differing noticeably from the theoretical $\pi/2$ for $n = 16384$ but in the favor of our algorithm.

2) M^3 CUDA Implementation: While highly optimizing the M^3 for Tensorflow was not an option, this section provides test results for a simple CUDA implementation of the multiplication (Fig. 2) of two random matrices with varying parameters. Kernels are currently optimized for power of two sizes, with a minimum size n of 256. A maximum matrix size of 16384 (1 GB per matrix) has been tested, due to GPU memory limitations. Results (single floating-point precision) are compared both to standard matrix multiplication and to $ASS^\top B$, where S is a sign matrix. Computing $ASS^\top B$ just requires three CUBLAS calls and therefore can be considered optimal and optimized.

Table I summarizes the factors affecting performance between the use of signed matrices [30] and the proposed method. While our technique requires more samples to estimate dot products at angles close to $\pi/2$ due to a higher variance, it also benefits from most operations being done by binary operators that can process 64 bits at a time (as implemented) to calculate the Hamming distance. However, a breakdown of the performance of the different components of the pipeline (Fig. 4) also shows that operations with a theoretically negligible complexity have a significant impact on performance, especially when the matrix size is small.

Fig.5 shows the performance gains obtained with our new technique and compare them to both a standard multiplication and Clarkson and Woodruff approach [30]. As expected, some significant speedup is obtained for large matrix multiplications, where our method is measured to be up to 4.64× as fast as the use of signed matrices [30] after correcting for the higher variance per iteration and as implemented. When compared to a standard matrix multiplication, we can be up to 10× as fast depending on the error required. For small matrix sizes, our algorithm is actually slower, which may not have a practical impact as most concerns with matrix multiplication performances arise from the use of large matrices.

3) Using the M^3 for Training: We study in Fig. 6 the effect of directly replacing the standard matrix multiplications with our M^3 version in the training of two DNN models with two and three dense layers of the MNIST models (no data augmentation) being replaced. The replacements are made in either the forward pass, the backward pass, or both passes. Unless stated otherwise, we use a batch size of 1024 to be representative of a real-world application as our algorithm can only replace matrix–matrix multiplications efficiently (speedwise), but not matrix–vector multiplications. This large batch size does not affect the convergence rate but may be impractical in some situations as this increases the memory requirements. The best results have been obtained by reusing the same random matrices (matrix E in Algorithm 2, one unique matrix per layer) in the forward passes, while these random matrices are always regenerated for every multiplication performed during the backpropagation phase. All other combinations have severely degraded performance, but we do not have currently an explanation for this.

Fig. 6 shows that both convergence rate and accuracy improve as we are using more planes for the approximation. The algorithm converges more or less closely to the reference model as expected. Combining both forward propagation and backpropagation passes also logically results in a lower accuracy than only using one of them. These are important results as training with k = 1024 requires approximately the same number of operations for calculating the Hamming distance as floating-point operations with the standard matrix multiplication to obtain similar results. The Hamming distance code can, however, process up to 64 operations per instruction as they operate on single bits. This performance may, however,

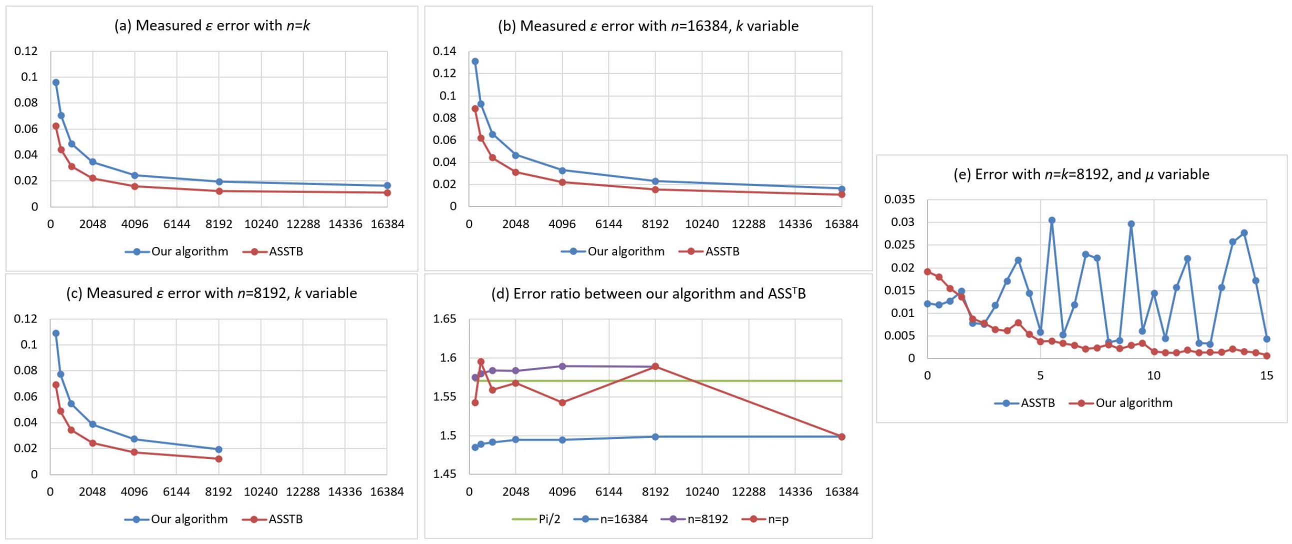


Fig. 3. Left: error comparison between our method and [30]. Results are given by either varying the sizes of matrix A and B with the number of samples p equal to n ($n = p$) or by fixing n and varying the number of samples used for the approximation. (e) Comparison of the final error obtained with our method and [30] after varying the μ parameter of the Gaussian RNG used (with $\sigma = 1$) when approximating AB. (a) Measured ε error with $n = k$. (b) Measured ε error with $n = 16384$, k variable. (c) Measured ε error with $n = 8192$, k variable. (d) Error ratio between our algorithm and ASS$^{\mathrm{T}}$B. (e) Error with $n = k = 8192$, and μ variable.

TABLE I

COMPARISON OF THE DIFFERENT PRACTICAL FACTORS INFLUENCING THE PERFORMANCE OF THE SIGNED MATRIX APPROXIMATION [30] AND OUR OWN HAMMING DISTANCE KERNEL (GPU: NVIDIA 1080TI)

Factor	Signed Matrices [30]	Our Approach
Theoretical Complexity - Square matrices	$O(n^2)$	$O(n^2)$
Base variance / Sampling constant	1 for all angles (Fig. 3e)	$\pi^2/4 \approx 2.46$ @ angles $\pi/2$ and $-\pi/2$ 0 @ angles 0 & π
Core arithmetic instructions per sample per entry	1 Fused Multiply and Add	1 PopCount, 1 XOR, 1 ADD (32 or 64 bits inst.)
Other performance factors	3 matrix multiplications needed. 6 floating-point operations per sample per entry.	3 bit operations per sample per entry. 32 or 64 bits processed per instruction. Extra operations of lower theoretical complexity taking a non negligible time (Fig. 5).
Peak throughput as measured	~10 TFlops	~82 TBits/s as implemented
Kernel implementations	CUDA libraries only (3 matrix mult.)	Own Hamming distance kernel (Fig. 2)

be overshadowed by the overhead created by lower complexity stages (e.g., the FFT stage). Fig. 5 tells us indeed that our current CUDA implementation of the M^3 would only perform at 0.4× the speed of a standard matrix multiplication and therefore be impractical in this example. Furthermore, our matrix multiplication may only be workable on shallow networks as the approximation error is likely to be amplified with the extra layers, as demonstrated by the impact of moving from two to three layers. Finally, we observe most often from the different graphs that the differences between the reference models and the approximated ones seem to be approximately halved every time one doubles the number of planes. Although not a proof at all, this would indicate a somehow surprising linear convergence to the true model. This may, however, not be in contradiction with the convergence rate of a Monte Carlo approximation as we are measuring the accuracy of the model and not that of the matrix multiplication.

C. Compression of FC Layers

We tested our Mediterranean diet on the FC layers of standard neural network models such as VGG16 and are mainly concerned in observing the effect of replacing the standard matrix multiplication in the pipeline of these models with our M^3. As such, the low error obtained in our tests, while satisfactory, may be beaten by other networks not using FC layers. We use VGG16 on three datasets (CIFAR10, CIFAR100 [47], and CINIC-10 [46]) and use two other standard models for MNIST [45]. In particular, VGG16 has three FC layers that represent 56% of the total size of the model when having 3×32^2 input images.

Details of the networks used are given in Table III. For training, we have used a weight decay of 5×10^{-4}, batch normalization, and dropout between CLs. We have used ReLU activation functions after each layer, except for the last layer where a softmax function has been used. In addition, data augmentation was created by rotating images 15° for CIFAR and CINIC datasets and 8° for MNIST. Furthermore, images were shifted by 10% randomly on both the x- and y-axes. Results with and without augmentation are given for MNIST. Experiments have been carried out with a varying number k of hyperplanes. The size of the compressed layers is calculated from the size of the bit representation of the layers, the norms

TABLE II

COMPARISON OF THE COMPRESSION RATE WITH BINARY-CONNECT [32] AND XNOR-NET [33]. OUR TECHNIQUE ALLOWS CHANGING THE COMPRESSION RATE TO FAVOR EITHER COMPRESSION OR QUALITY

Dataset	MNIST		CIFAR10		
Technique	Accuracy	size	Accuracy	size	FC-layers only size
Binary-Connect	98.78%	0.36 MB (3.26%)	90.10%	**1.67 MB (3.13%)**	1.13 MB (3.13%)
XNOR-Net	97.65%	0.38 MB (3.40%)	91.24%	1.68 MB (3.15%)	1.14 MB (3.15%)
M^3, $k = 2048$	**99.37%**	0.50 MB (7.24%)	**91.81%**	17.97 MB (33.59%)	0.51 MB (1.43%)
M^3, $k = 1024$	98.61%	0.43 MB (3.87%)	91.29%	17.72 MB (33.12%)	0.26 MB (0.73%)
M^3, $k = 512$	97.92%	**0.24 MB (2.18%)**	90.88%	17.59 MB (32.88%)	**0.14 MB (0.38%)**

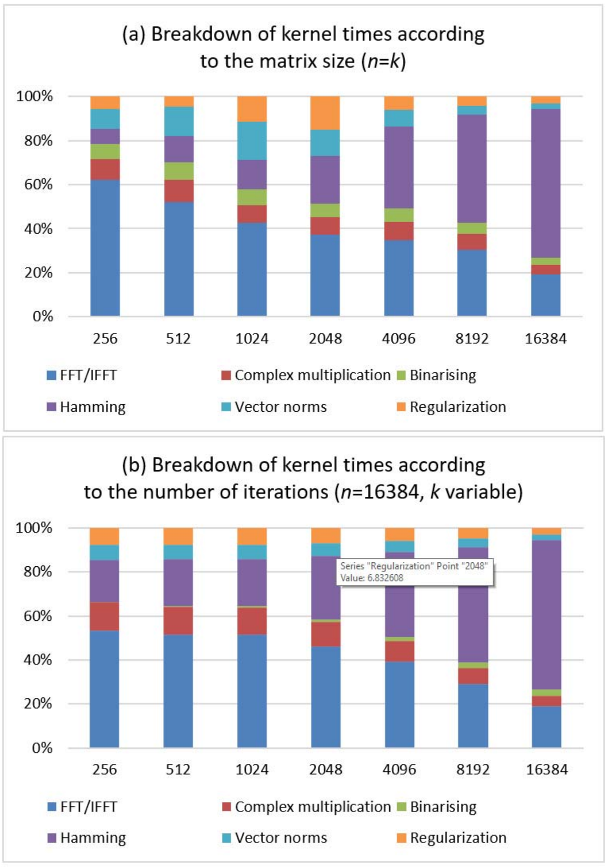
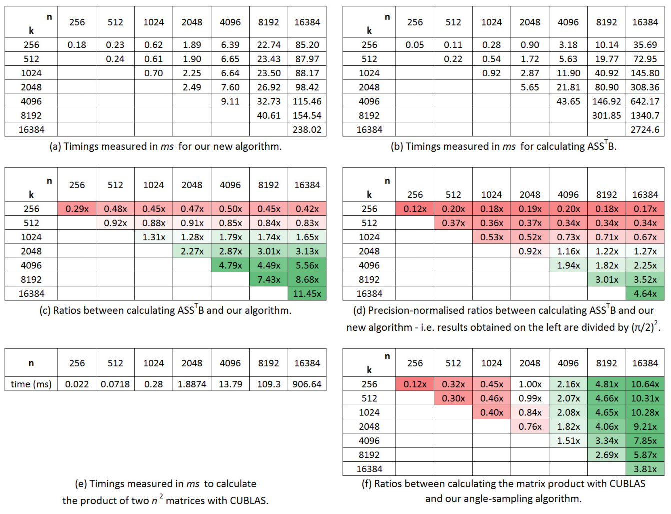


Fig. 4. Breakdown of performance for the various kernels used in our GPU implementation. (a) Breakdown of kernel times according to the matrix size ($n = k$). (b) Breakdown of kernel times according to the number of iterations ($n = 16384$, k variable).



k \ n	256	512	1024	2048	4096	8192	16384
256	0.18	0.23	0.62	1.89	6.39	22.74	85.20
512		0.24	0.61	1.90	6.65	23.43	87.97
1024			0.70	2.25	6.64	23.50	88.17
2048				2.49	7.60	26.92	98.42
4096					9.11	32.73	115.46
8192						40.61	154.54
16384							238.02

(a) Timings measured in *ms* for our new algorithm.

k \ n	256	512	1024	2048	4096	8192	16384
256	0.05	0.11	0.28	0.90	3.18	10.14	35.69
512		0.22	0.54	1.72	5.63	19.77	72.95
1024			0.92	2.87	11.90	40.92	145.80
2048				5.65	21.81	80.90	308.36
4096					43.65	146.92	642.17
8192						301.85	1340.7
16384							2724.6

(b) Timings measured in *ms* for calculating ASS^{T}B.

k \ n	256	512	1024	2048	4096	8192	16384
256	0.29x	0.48x	0.45x	0.47x	0.50x	0.45x	0.42x
512		0.92x	0.88x	0.91x	0.85x	0.84x	0.83x
1024			1.31x	1.28x	1.79x	1.74x	1.65x
2048				2.27x	2.87x	3.01x	3.13x
4096					4.79x	4.49x	5.56x
8192						7.43x	8.68x
16384							11.45x

(c) Ratios between calculating ASS^{T}B and our algorithm.

k \ n	256	512	1024	2048	4096	8192	16384
256	0.12x	0.20x	0.18x	0.19x	0.20x	0.18x	0.17x
512		0.37x	0.36x	0.37x	0.34x	0.34x	0.34x
1024			0.53x	0.52x	0.73x	0.71x	0.67x
2048				0.92x	1.16x	1.22x	1.27x
4096					1.94x	1.82x	2.25x
8192						3.01x	3.52x
16384							4.64x

(d) Precision-normalised ratios between calculating ASS^{T}B and our new algorithm - i.e. results obtained on the left are divided by $(\pi/2)^2$.

n	256	512	1024	2048	4096	8192	16384
time (ms)	0.022	0.0718	0.28	1.8874	13.79	109.3	906.64

(e) Timings measured in *ms* to calculate the product of two n^2 matrices with CUBLAS.

k \ n	256	512	1024	2048	4096	8192	16384
256	0.12x	0.32x	0.45x	1.00x	2.16x	4.81x	10.64x
512		0.30x	0.46x	0.99x	2.07x	4.66x	10.31x
1024			0.40x	0.84x	2.08x	4.65x	10.28x
2048				0.76x	1.82x	4.06x	9.21x
4096					1.51x	3.34x	7.85x
8192						2.69x	5.87x
16384							3.81x

(f) Ratios between calculating the matrix product with CUBLAS and our angle-sampling algorithm.

Fig. 5. Performance comparison between our method, ASS$^T B$ [30] (b, c, and d, CUBLAS implementation), and a direct matrix multiplication of A and B (e and f, CUBLAS implementation). (a) Timings measured in ms for our new algorithm. (b) Timings measured in ms for calculating ASS$^{\text{T}}$B. (c) Ratio between calculating ASS$^{\text{T}}$B and our algorithm. (d) Precision-normalised ratios between calculating ASS$^{\text{T}}$B and our new algorithm - i.e. results obtained on the left are divided by $(\Pi/2)^2$. (e) Timings measured in ms to calculate the product of two n^2 matrices with CUBLAS. (f) Ratios between calculating the matrix product with CUBLAS and our angle-sampling algorithm.

$||W_i^l||$ of the weight matrix rows, and the seed value that is negligible. As our random numbers are calculated from the seed value, they do not need to be stored. We, however, include the storage requirements for the tests where random numbers are further obtained by rotation of columns of E (i.e., Toeplitz case) as this may have some practical implications, including removing the need of calculating these random numbers on the fly when inferring. Table III shows the results for the four datasets and Fig. 7 shows the convergence to the original network according to the number of planes.

All experiments on VGG16 showed that the FC layers can be compressed to around 1% of their original size without any meaningful loss of accuracy. The compression rate for the MNIST models is also very satisfactory at around 50× without a significant degradation of accuracy, although lower than VGG16 results in general. We conjecture that as VGG16 FC layers are larger, they become easier to compress. It, however, looks like more hyperplanes are needed for the augmented MNIST dataset compared to the original one. While the compression rates are still very good, we still need two to four times more space to see no difference with the original network. We do not know at this stage if this result can be improved upon, by, for instance, improving the random number sequences.

Obviously, compressing the FC layers translates into a significant reduction in the size of these models. For the MNIST models, as all the layers are FC layers, the model can easily be compressed up to 25× without any loss of accuracy. Also, while the original VGG16 network size was mostly dictated by the FC layers, the Mediterranean diet just makes them negligible in size, leading to a compression of VGG16 by more than 2. Finally, storing the random numbers does not have a big impact on the memory footprint but still allows faster implementations less relying on floating-point calculations as this part of the pipeline can be now ensured by a fast convolution performed in the Fourier space.

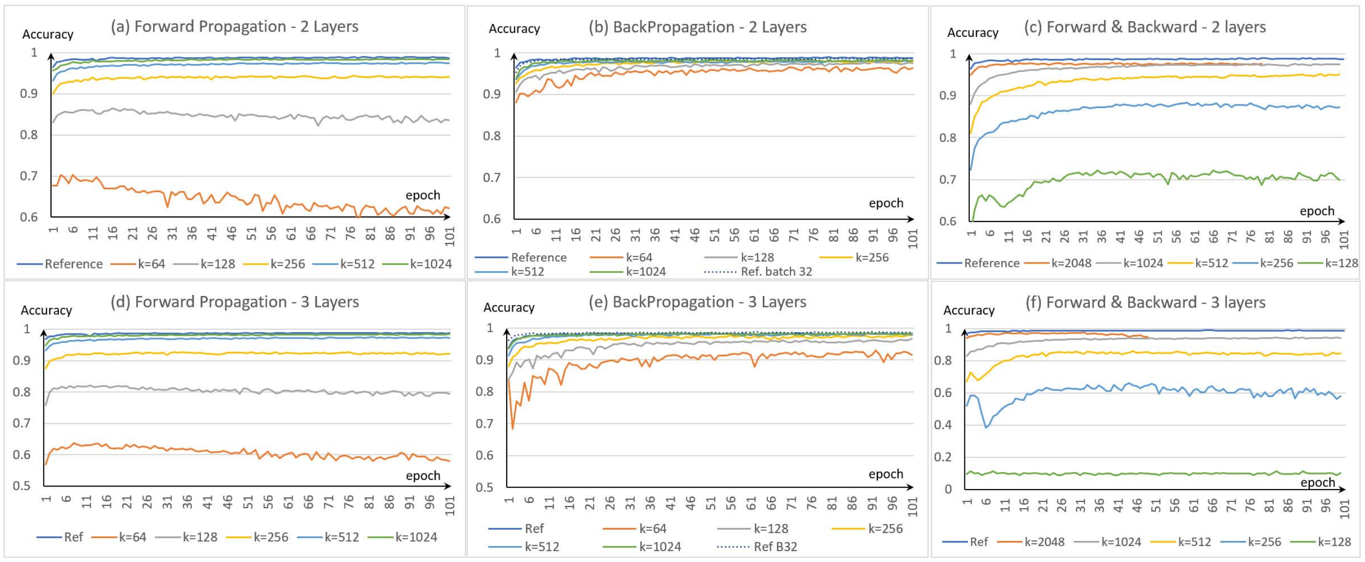


Fig. 6. Effect of replacing the standard matrix multiplication with the M^3 version for training the MNIST datasets in all but the last FC layer. The horizontal axis represents the number of epochs, while the vertical axis represents the accuracy obtained on the testing dataset. We study backward-only, forward-only, and backward–forward replacement cases. The number of planes k used is the same for each layer and each pass in a single experiment. Unless stated otherwise (B32), the batch size used is 1024. Results show that training accuracy (test dataset) increases with the number of plane and also decreases when adding an extra layer. (a) Forward propagation—two layers. (b) Backpropagation—two layers. (c) Forward and back—two layers. (d) Forward propagation—three layers. (e) Backpropagation—three layers. (f) Forward and back—three layers.

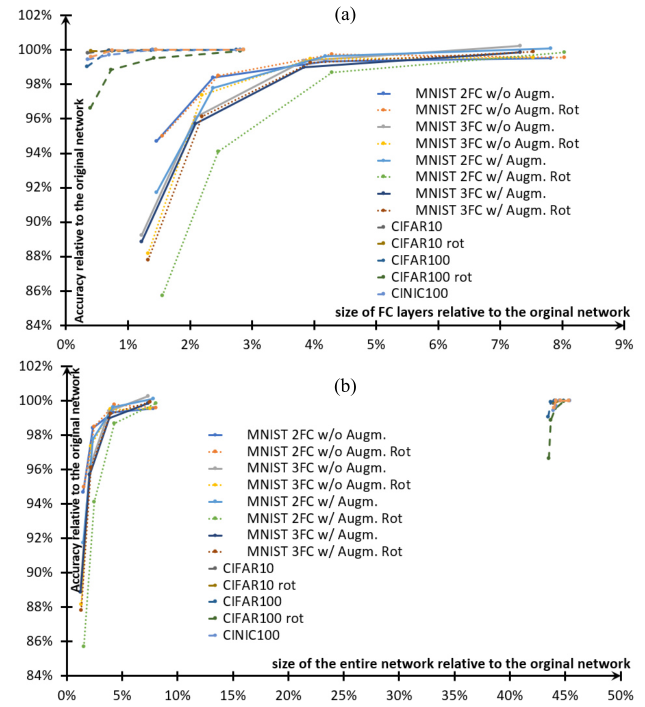


Fig. 7. Convergence of accuracy according to the number of hyperplanes used. The horizontal axis displays the compression obtained for (a) internal FC layers and (b) neural network as a whole (full network). The vertical axis displays the obtained accuracy of the network relative to that of the original, unmodified network—with 100% meaning that the compressed network has the same accuracy as the original one. All network models have been tested with 256, 512, 1024, and 2048 hyperplanes as represented with continuous or dashed lines.

We finally compare the compression rates we obtained (cf. Table II) with two other techniques that are Binary-Connect [32] and XNOR-Net [33]. The table shows that similar levels of compression and accuracy can be obtained by the three techniques for FC layers. While Binary-Connect and XNOR-Net only handle a single bit of information, our approach is more flexible as it allows to choose any random number of planes we wish and therefore allows parameterizing compression levels. However, we do not propose yet a way to compress convolutional networks, and therefore, the two techniques cited perform much better overall in CIFAR10. The similarities in the results make us believe that these two methods may possibly implicitly learn the M^3 pipeline while training.

VI. Conclusion

This article has first proposed and analyzed the M^3, a new, fairly simple, unbiased algorithm for approximating matrix multiplications. While the theoretical bound obtained is optimal and similar to the currently best known randomized methods [30], it does offer increased convergence when dealing with nonorthogonal rows and columns. It is also amenable to various bitwise optimizations that accelerate computations greatly as the cost of combined XOR/POPCOUNT units could be significantly lower than that of FMA units in terms of area and energy consumption.

This article shows that this new algorithm can be used in machine learning, for instance, to train the FC layers of a neural network. We have also demonstrated a Mediterranean diet algorithm for FC layers, allowing a compression rate for these layers as high as 100× in the case of VGG16 and so without a drop of accuracy. This result is similar or better to similar techniques published in the area, with the extended benefit that most inferring operations can be performed on a binary stream using a combination of XOR and POPCOUNT operators. This should allow simplifying the architecture of embedded inference processors and, as such,

TABLE III

COMPRESSION RATES OF FC LAYERS AND OVERALL NETWORK ACCORDING TO THE NUMBER k OF HYPERPLANES CHOSEN AND WHETHER ROTATION OF RANDOM NUMBERS HAS BEEN USED (ROT). THE MODIFIED/COMPRESSED LAYERS ARE IN BOLD. **FC**: FC LAYER. BN: BATCH NORMALIZATION LAYER. RELU: RECTIFIED LINEAR UNIT LAYER. SFMX: SOFTMAX LAYER

(a) MNIST without Data Augmentation
28×28:DropOut:**FC1024**:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:FC10:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	99.02%	7.11 MB (100%)	7.11 MB (100%)
k = 256	95.51%	0.11 MB (1.48%)	0.11 MB (1.48%)
k = 512	98.01%	0.18 MB (2.47%)	0.18 MB (2.47%)
k = 1024	98.61%	0.30 MB (4.23%)	0.30 MB (4.23%)
k = 2048	98.90%	0.55 MB (7.75%)	0.55 MB (7.75%)
k = 256 (rot)	94.06%	0.11 MB (1.59%)	0.11 MB (1.59%)
k = 512 (rot)	97.55%	0.18 MB (2.58%)	0.18 MB (2.58%)
k = 1024 (rot)	98.78%	0.31 MB (4.34%)	0.31 MB (4.34%)
k = 2048 (rot)	98.82%	0.57 MB (7.97%)	0.57 MB (7.97%)

(b) MNIST without Data Augmentation
28×28:DropOut:**FC1024**:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:FC10:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	99.12%	11.1 MB (100%)	11.1 MB (100%)
k = 256	93.76%	0.14 MB (1.23%)	0.14 MB (1.23%)
k = 512	97.92%	0.24 MB (2.18%)	0.24 MB (2.18%)
k = 1024	98.61%	0.43 MB (3.87%)	0.43 MB (3.87%)
k = 2048	99.37%	0.80 MB (7.25%)	0.80 MB (7.25%)
k = 256 (rot)	94.06%	0.15 MB (1.34%)	0.15 MB (1.34%)
k = 512 (rot)	97.54%	0.25 MB (2.29%)	0.25 MB (2.29%)
k = 1024 (rot)	98.78%	0.44 MB (3.97%)	0.44 MB (3.97%)
k = 2048 (rot)	98.72%	0.83 MB (7.46%)	0.83 MB (7.46%)

(c) MNIST with Data Augmentation
28×28:DropOut:**FC1024**:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:FC10:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	99.29%	7.11 MB (100%)	7.11 MB (100%)
k = 256	91.08%	0.11 MB (1.48%)	0.11 MB (1.48%)
k = 512	97.08%	0.18 MB (2.47%)	0.18 MB (2.47%)
k = 1024	98.92%	0.30 MB (4.23%)	0.30 MB (4.23%)
k = 2048	99.38%	0.55 MB (7.75%)	0.55 MB (7.75%)
k = 256 (rot)	85.10%	0.11 MB (1.59%)	0.11 MB (1.59%)
k = 512 (rot)	93.43%	0.18 MB (2.58%)	0.18 MB (2.58%)
k = 1024 (rot)	97.98%	0.31 MB (4.34%)	0.31 MB (4.34%)
k = 2048 (rot)	99.14%	0.57 MB (7.97%)	0.57 MB (7.97%)

(d) MNIST with Data Augmentation
28×28:DropOut:**FC1024**:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:FC10:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	99.52%	11.1 MB (100%)	11.1 MB (100%)
k = 256	88.43%	0.14 MB (1.23%)	0.14 MB (1.23%)
k = 512	95.25%	0.24 MB (2.18%)	0.24 MB (2.18%)
k = 1024	98.51%	0.43 MB (3.87%)	0.43 MB (3.87%)
k = 2048	99.37%	0.80 MB (7.25%)	0.80 MB (7.25%)
k = 256 (rot)	87.38%	0.15 MB (1.34%)	0.15 MB (1.34%)
k = 512 (rot)	95.66%	0.25 MB (2.29%)	0.25 MB (2.29%)
k = 1024 (rot)	98.78%	0.44 MB (3.97%)	0.44 MB (3.97%)
k = 2048 (rot)	99.43%	0.83 MB (7.46%)	0.83 MB (7.46%)

(e) CIFAR10
32×32×3:(VGG16 [48] Convolutional Part):
FC1024:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:**FC10**:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	93.03%	72.2 MB (100%)	128.42 MB (100%)
k = 256	92.88%	0.25 MB (0.35%)	56.39 MB (43.95%)
k = 512	93.01%	0.51 MB (0.70%)	56.64 MB (44.15%)
k = 1024	93.03%	1.01 MB (1.40%)	57.15 MB (44.54%)
k = 2048	93.03%	2.03 MB (2.81%)	58.30 MB (45.39%)
k = 256 (rot)	92.96%	0.29 MB (0.40%)	56.42 MB (43.98%)
k = 512 (rot)	92.96%	0.54 MB (0.75%)	56.67 MB (44.18%)
k = 1024 (rot)	93.03%	1.05 MB (1.45%)	57.18 MB (44.57%)
k = 2048 (rot)	93.03%	2.06 MB (2.86%)	58.20 MB (45.36%)

(f) CIFAR100
32×32×3:(VGG16 [48] Convolutional Part):
FC1024:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:**FC100**:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	71.84%	73.6 MB (100%)	129.7 MB (100%)
k = 256	71.15%	0.26 MB (0.35%)	56.39 MB (43.48%)
k = 512	71.79%	0.51 MB (0.70%)	56.64 MB (43.67%)
k = 1024	71.83%	1.02 MB (1.39%)	57.16 MB (44.07%)
k = 2048	71.84%	2.05 MB (2.78%)	58.18 MB (44.86%)
k = 256 (rot)	69.42%	0.29 MB (0.39%)	56.42 MB (43.50%)
k = 512 (rot)	71.02%	0.54 MB (0.73%)	56.67 MB (43.70%)
k = 1024 (rot)	71.50%	1.05 MB (1.42%)	57.18 MB (44.09%)
k = 2048 (rot)	71.80%	2.06 MB (2.81%)	58.20 MB (44.87%)

(g) CINIC10
32×32×3:(VGG16 [48] Convolutional Part):
FC1024:BN:ReLU:DropOut:**FC1024**:BN:ReLU:DropOut:**FC10**:SFMX

#Hyperplanes	Accuracy	Size of FC layers(%)	Total Network Size (%)
Original Net.	79.25%	72.2 MB (100%)	128.42 MB(100%)
k = 256	78.81%	0.25 MB (0.35%)	56.39 MB(43.95%)
k = 512	79.02%	0.51 MB (0.70%)	56.64 MB (44.15%)
k = 1024	79.25%	1.01 MB (1.40%)	57.15 MB(44.54%)
k = 2048	79.25%	2.03 MB (2.81%)	58.30 MB (45.39%)
k = 256 (rot)	78.92%	0.29 MB (0.40%)	56.42 MB (43.98%)
k = 512 (rot)	79.24%	0.54 MB (0.75%)	56.67 MB (44.18%)
k = 1024 (rot)	79.25%	1.05 MB (1.45%)	57.18 MB (44.57%)
k = 2048 (rot)	79.25%	2.06 MB (2.86%)	58.20 MB (45.36%)

reduce significantly their energy consumption. Our CUDA implementation indeed demonstrates that a significant speedup can be obtained for large matrix sizes based mainly on the sole use of these operators.

Overall, the Mediterranean diet is a technique closely related to recent research done in the area of random projections, low-rank approximations, and binary nets, and combining all the benefits of these methods into one framework. Further investigations may include tweaking the RNG, compressing other types of layers and networks, studying training scalability for larger networks, or even accelerating other scientific problems.

REFERENCES

[1] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, 1969, doi: 10.1007/BF02165411.

[2] P. Drineas and M. W. Mahoney, "RandNLA: Randomized numerical linear algebra," *Commun. ACM*, vol. 59, no. 6, pp. 80–90, May 2016, doi: 10.1145/2842602.

[3] D. Coppersmith, "Rapid multiplication of rectangular matrices," *SIAM J. Comput.*, vol. 11, no. 3, pp. 467–471, Aug. 1982.

[4] R. W. Brockett and D. Dobkin, "On the number of multiplications required for matrix multiplication," *SIAM J. Comput.*, vol. 5, no. 4, pp. 624–628, Dec. 1976.

[5] D. Coppersmith, "Rectangular matrix multiplication revisited," *J. Complex.*, vol. 13, no. 1, pp. 42–49, Mar. 1997.

[6] V. Y. Pan, "Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations," in *Proc. 19th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1978, pp. 166–176, doi: 10.1109/SFCS.1978.34.

[7] D. Bini, M. Capovani, F. Romani, and G. Lotti, "$O(n^{2.7799})$ complexity for n*n approximate matrix multiplication," *Inf. Process. Lett.*, vol. 8, pp. 234–235, Jun. 1979.

[8] A. Schönhage, "Partial and total matrix multiplication," *SIAM J. Comput.*, vol. 10, no. 3, pp. 434–455, Aug. 1981.

[9] F. Romani, "Some properties of disjoint sums of tensors related to matrix multiplication," *SIAM J. Comput.*, vol. 11, no. 2, pp. 263–267, May 1982.

[10] D. Coppersmith and S. Winograd, "On the asymptotic complexity of matrix multiplication," *SIAM J. Comput.*, vol. 11, no. 3, pp. 472–492, Aug. 1982.

[11] V. Strassen, "Relative bilinear complexity and matrix multiplication," *J. für Die Reine Und Angewandte Mathematik*, vol. 1987, nos. 375–376, pp. 406–443, 1987.

[12] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *J. Symbolic Comput.*, vol. 9, no. 3, pp. 251–280, Mar. 1990, doi: 10.1016/S0747-7171(08)80013-2.

[13] A. J. Stothers, *On the Complexity of Matrix Multiplication*. Edinburgh, U.K.: Univ. of Edinburgh, 2010. [Online]. Available: https://books.google.co.U.K./books?id=J1KEkgEACAAJ

[14] V. V. Williams. (2014). *Multiplying Matrices in* $\upsilon O(n^{2.373})$ *Time*. [Online]. Available: https://people.csail.mit.edu/virgi/matrixmult-f.pdf

[15] F. Le Gall, "Powers of tensors and fast matrix multiplication," in *Proc. 39th Int. Symp. Symbolic Algebr. Comput. (ISSAC)*, 2014, pp. 296–303, doi: 10.1145/2608628.2608664.

[16] F. Le Gall, "Faster algorithms for rectangular matrix multiplication," in *Proc. IEEE 53rd Annu. Symp. Found. Comput. Sci.*, Oct. 2012, pp. 514–523.

[17] J. Hopcroft and J. Musinski, "Duality applied to the complexity of matrix multiplications and other bilinear forms," in *Proc. 5th Annu. ACM Symp. Theory Comput.*, 1973, pp. 73–87.

[18] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev, "On economical construction of the transitive closure of a directed graph," *Sov. Math. Doklady*, vol. 11, no. 5, pp. 1209–1210, 1970.

[19] N. Bansal and R. Williams, "Regularity lemmas and combinatorial algorithms," in *Proc. 50th Annu. IEEE Symp. Found. Comput. Sci.*, Oct. 2009, pp. 745–754.

[20] T. M. Chan, "Speeding up the four russians algorithm by about one more logarithmic factor," in *Proc. 26th Annu. ACM-SIAM Symp. Discrete Algorithms*, Oct. 2015, pp. 212–217.

[21] H. Yu, "An improved combinatorial algorithm for Boolean matrix multiplication," *Int. Colloq. Automata, Lang. Program.*, vol. 261, pp. 1094–1105, Aug. 2015.

[22] J. Wiedermann, "Fast nondeterministic matrix multiplication via derandomization of freivalds' algorithm," in *Proc. Int. Conf. Theor. Comput. Sci.* Berlin, Germany: Springer, 2014, pp. 123–135.

[23] A. Lingas and D. Sledneu, "Vector convolution in $O(n)$ steps and matrix multiplication in $O(n^2)$ steps," in *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 21. 2014, no. 39, pp. 1–6. [Online]. Available: https://eccc.weizmann.ac.il/report/2014/039/

[24] I. Korec and J. Wiedermann, "Deterministic verification of integer matrix multiplication in quadratic time," in *Proc. SOFSEM Int. Conf. Current Trends Theory Pract. Inform.* Cham, Switzerland: Springer, 2014, pp. 375–382.

[25] Z. Bar-Yossef, "Sampling lower bounds via information theory," in *Proc. 35th ACM Symp. Theory Comput. (STOC)*, 2003, pp. 335–344.

[26] T. Sarlos, "Improved approximation algorithms for large matrices via random projections," in *Proc. 47th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*, Oct. 2006, pp. 143–152.

[27] P. Drineas and R. Kannan, "Fast Monte-Carlo algorithms for approximate matrix multiplication," in *Proc. 42nd IEEE Symp. Found. Comput. Sci.*, 2001, pp. 452–459.

[28] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast Monte Carlo algorithms for matrices I: Approximating matrix multiplication," *SIAM J. Comput.*, vol. 36, no. 1, pp. 132–157, Jan. 2006.

[29] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Contemp. Math.*, vol. 26, Jan. 1984.

[30] K. L. Clarkson and D. P. Woodruff, "Numerical linear algebra in the streaming model," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2009, pp. 205–214, doi: 10.1145/1536414.1536445.

[31] D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*.

[32] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," 2015, *arXiv:1511.00363*.

[33] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," 2016, *arXiv:1603.05279*.

[34] J. Chen, Y. Liu, H. Zhang, S. Hou, and J. Yang, "Propagating asymptotic-estimated gradients for low bitwidth quantized neural networks," *IEEE J. Sel. Topics Signal Process.*, vol. 14, no. 4, pp. 848–859, May 2020.

[35] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," 2017, *arXiv:1710.09282*.

[36] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, "Learning separable filters," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2013, pp. 2754–2761.

[37] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," 2013, *arXiv:1306.0543*.

[38] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," 2014, *arXiv:1405.3866*.

[39] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 28, 2015, pp. 1–9.

[40] A. Rahimi and B. Recht, "Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, pp. 1313–1320.

[41] Q. Viet Le, T. Sarlos, and A. J. Smola, "Fastfood: Approximate kernel expansions in loglinear time," 2014, *arXiv:1408.3060*.

[42] Z. Yang *et al.*, "Deep fried convnets," 2014, *arXiv:1412.7149*.

[43] M. X. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *J. Assoc. Comput. Mach.*, vol. 42, no. 6, pp. 1115–1145, 1995, doi: 10.1145/227683.227684.

[44] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proc. 34th Annu. ACM Symp. Theory Comput. (STOC)*, 2002, pp. 380–388, doi: 10.1145/509907.509965.

[45] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[46] L. N. Darlow, E. J. Crowley, A. Antoniou, and A. J. Storkey, "CINIC-10 is not ImageNet or CIFAR-10," 2018, *arXiv:1810.03505*.

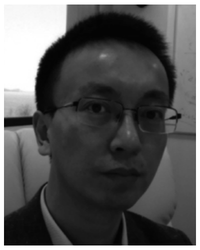
[47] A. Krizhevsky, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep., 2012. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf and https://scholar.google.com/scholar?hl=en&q=Learning+Multiple+Layers+of+Features+from+Tiny+Images

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

Hassan Eshkiki received the B.Sc. degree from University of Applied Science and Technology, Fouman, Iran, in 2010, the M.Sc. degree in advanced information systems and software engineering from Grenoble University, Grenoble, France, in 2019, and the M.Sc. degree in advanced computer science from Swansea University, Swansea, U.K., in 2019, where he is currently pursuing the Ph.D. degree, with a focus on new algorithms for machine learning and explainable artificial intelligence.

Benjamin Mora received the D.E.A. (M.Sc. equivalent) and Ph.D. degrees in computer science from Paul Sabatier University, Toulouse, France, in 1998 and 2001, respectively.

In 2004, he joined Swansea University, where he is currently an Associate Professor in computer science. His primary research interests include general algorithms for computer graphics, visualization, and machine learning.



Xianghua Xie (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from the University of Bristol, Bristol, U.K., in 2002 and 2006, respectively.

He is currently a Full Professor with the Department of Computer Science, Swansea University, Swansea, U.K., where he is leading the Computer Vision and Machine Learning Laboratory, Swansea University. He has published more than 160 refereed conference and journal publications and (co-)edited several conference proceedings. His research interests include various aspects of pattern recognition and machine intelligence and their applications to real-world problems.

Dr. Xie is a member of British Machine Vision Association (BMVA). He is an Associate Editor of a number of journals, including *Pattern Recognition* and *IET Computer Vision*.