Cooking in the Dark: Performance Evaluation of Matrix-Vector Operations in Neural Networks under Homomorphic Encryption

Lukas Horn[1], Lennart Svoboda[1], Simon Engel[1], Thomas Prantl[1], and Samuel Kounev[1]

[1]University of Würzburg, Würzburg, Germany

Abstract

Homomorphic Encryption (HE) is an emerging technology that enables Privacy-Preserving Machine Learning (PPML). By providing the capability to compute on encrypted data, it facilitates the evaluation of Neural Networks (NNs) in client-server scenarios, where sensitive data is sent to an untrusted server for inference. As NNs rely heavily on matrix–vector multiplications, we evaluate several naive methods for implementing these operations within the constraints of HE. We demonstrate that the diagonal method is a highly versatile approach and the fastest overall. However, the traditional row-wise method offers the lowest memory footprint, making it suitable for memory-constrained environments.

1 Introduction

NNs have moved quickly from research to core infrastructure, and run very efficiently using the parallel computing powers of Graphics Processing Units (GPUs). Matrix operations sit at the heart of almost every layer, and decades of engineering have optimized their execution. Furthermore, modern cloud platforms eliminate the need to purchase or maintain expensive hardware directly, instead providing compute on demand. However, this convenience often comes at the cost of privacy. Running inference on external hardware forces users to hand over sensitive data in plaintext. HE, and in particular the scheme Cheon-Kim-Kim-Song (CKKS) [5], addresses this by enabling direct computation on encrypted real numbers. Yet, CKKS is not a drop-in replacement for standard maths since it natively supports only addition and multiplication and is built on vectorization by packing values into slots that are then encrypted. Accessing individual elements requires cryptographic rotation of the vector. Every multiplication depletes a pre-defined budget for this operation. If that is used up, it is possible to restore most of it using so-called "bootstrapping". However, both operations are computationally expensive. Performance is therefore no longer driven by traditional Floating Point Operations Per Second (FLOPs), but by multiplicative depth and rotation count. While plaintext matrix operations are thoroughly understood, a similarly clear understanding is largely missing under CKKS. This paper addresses this gap by answering the research question of how naive matrix–vector multiplication methods (`Element`-wise, `Row`-wise, `Column`-wise, and `Diagonal`-wise) compare regarding execution time and memory usage.

2 Background & Limitations of HE

We refer to unencrypted data as "plaintext" and to encrypted data as "ciphertext". Dedicated keys are used to map elements between these domains.

The primary purpose of classical cryptography is to protect data in transit or at rest and any modification of encrypted data leads to failures or incorrect results.

Homomorphic Encryption (HE) differs from this paradigm by enabling the evaluation of operations directly on ciphertexts. For example, if two plaintexts p_1 and p_2 are encrypted into ciphertexts c_1 and c_2, they can be added to form $c' = c_1 + c_2$. Upon decrypting c', the resulting plaintext is $p' = p_1 + p_2$. This capability allows a computationally constrained client to outsource calculations on sensitive data to a server, preserving confidentiality by transmitting encrypted inputs and receiving encrypted results.

However, this technology also exhibits several limitations. In addition to a restricted set of operations and a limited multiplication depth, execution times are several orders of magnitude slower [2]. The CKKS scheme, in particular, relies on vectorization. While this offers Single Instruction, Multiple Data (SIMD) capabilities, it also introduces new restrictions. Rotating the elements in the encrypted vector requires a dedicated key and is a slow operation.

3 Matrix-Vector Algorithms

We leverage the aforementioned SIMD property to evaluate the performance of four matrix-vector multiplication algorithms applicable within a NN. Let the matrix be $A \in \mathbb{R}^{m \times n}$, where $m \in \mathbb{N}$ is the number of rows, $n \in \mathbb{N}$ is the number of columns, and a_{ij} denotes the j-th element in the i-th row. The input vector is defined as $v \in \mathbb{R}^n$. For our evaluation, we consider the following four algorithms that evaluate Av:

1) `Element`-wise: A baseline approach that does not utilize the SIMD property; instead, it iterates over individual row elements, multiplying them by

Author's Version

the corresponding vector elements and summing the result. 2) `Row`-wise: The schoolbook approach, which multiplies each row $a_{i_}$ by v and stores the sum in the i-th slot of the result. 3) `Column`-wise: Multiplies each column $a_{_j}$ by a vector uniformly filled with v_j. The final result is the sum of these vectors. 4) `Diagonal`-wise [6]: All diagonals of the matrix are extracted and subsequently multiplied by v. To ensure correctness, the second diagonal is multiplied by a left-rotated version of v, the third by a version left-rotated by two positions, and so forth.

4 Evaluation Environment

The neural network employed for this evaluation is a simple feedforward Multi-Layer Perceptron (MLP). It comprises eight input nodes that map to a hidden layer of four nodes with an activation function, followed by an output layer consisting of two nodes. These dimensions derive from the energy efficiency dataset [1] utilized in our experiments. This dataset features eight input variables mapped to two target variables (heating load and cooling load). We adapt this into a binary classification problem for house energy efficiency. The class is positive (1) if the combined load exceeds the median; otherwise, it is negative (0).

To determine the best-performing hidden layer size, we systematically experimented with sizes ranging from 3 to 7, finding that a size of 4 provided the best results. We applied an 80/20 split for the training and test sets, achieving an accuracy of 99.4 %. All experiments were executed on an Ubuntu 24.04 LTS machine equipped with an Intel(R) Xeon(R) CPU E5-2640 @ 2.60 GHz and 32 GB of RAM. For our implementation we use the OpenFHE library (v1.5.1)[1].We evaluated two variants for the activation function, specifically the ReLU and Sigmoid function, where each of them was approximated using two approaches. One is taken from Engel et al. [3] and the other one being the Chebyshev polynomials that are widely adopted in the HE context. This accounts for four different configurations in total. As Horn [4] demonstrated, only a rough approximation is required to maintain good network performance. Consequently, we set the approximation parameters in a way that the network achieves meaningful results, but the approximation function only roughly follows the exact function. Each scenario exhibits distinct characteristics, which is why we selected multiple variants to demonstrate that our findings regarding the multiplication methods hold consistently across all of them. For each configuration, we performed inference on every sample from the test set ($n = 154$). This yielded a total of 616 data points for the subsequent evaluation.[2]

[1] `https://github.com/openfheorg/openfhe-development/releases/tag/v1.5.1`

[2] The implementation, raw data, and plotting scripts, together with an extended evaluation, are provided in [7].

5 Evaluation

We begin by evaluating the runtime of the different algorithms, as shown in Figure 1a. Although our network utilizes these algorithms in two layers (first → hidden and hidden → output), we focus exclusively on the first one since the second exhibits similar behaviour, albeit with smaller values due to the reduced matrix dimensions. For the first layer a 4×8 matrix is multiplied by a 8×1 vector. In the following we report the mean value over all samples together with the standard derivation. All results are on the order of seconds; however, there are significant differences between them. The `Element`-wise method is the slowest, taking (47.89 ± 6.94) s. This is followed by the `Column` method at (25.62 ± 2.48) s and the `Row` method at (15.92 ± 2.58) s. The diagonal method is the fastest, requiring only (8.96 ± 1.00) s.

The reasons for these differences are twofold. The first is the dependency on the number of required rotations, which is one of the most computationally expensive operations in HE. The `Element` algorithm must rotate each element to the first index for multiplication and subsequently rotate the results back to their final indices. This process not only requires 12 rotation keys (one for each column and row) but is also highly inefficient, rendering the method 5.53 times slower than the `Diagonal` approach. In contrast, the `Diagonal` method requires only 7 keys and rotations.

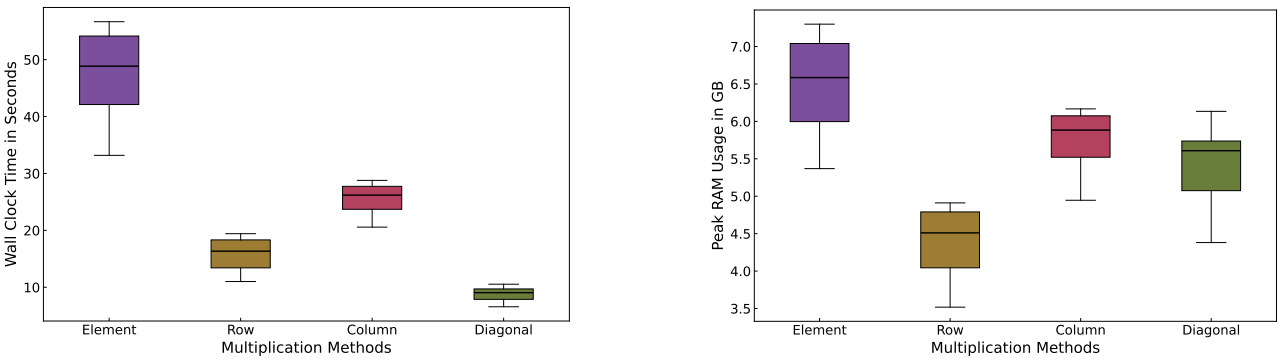
The second reason lies in the algorithmic design itself. The `Column` method requires only 6 keys—fewer than the `Diagonal` method—yet it is 2.86 times slower. This discrepancy stems from its replication procedure. To realize the multiplication, each individual element is replicated across a full vector and then multiplied by the corresponding column. The replication itself is computationally inexpensive, requiring only a logarithmic number of multiplications, and similar operations are utilized by the other algorithms. The key difference, however, is that the `Column` method requires this replication for every element (eight here). This accounts for the majority of the runtime.

Therefore, we conclude that the `Diagonal` method is the most efficient multiplication technique in terms of runtime. Given that the alternative methods are at least twice as slow, their use is difficult to justify in scenarios where low latency is a critical requirement.

Next, we analyse the peak memory usage that was collected by measuring the maximum Resident Set Size (RSS) of the process after the first multiplication. It is depicted in Figure 1b.

Here, the trend differs from that of the runtime. The `Element`-wise algorithm remains the most resource-intensive at (6.48 ± 0.73) GB, followed by the `Column` method at (5.73 ± 0.47) GB. However, the relative performance of the most efficient methods is reversed: the `Diagonal` method is now only the second most efficient, consuming (5.53 ± 0.51) GB, while the `Row` method performs best, requiring just (4.42 ± 0.47) GB. Given the

Author's Version



(a) Wall Clock Time

(b) Peak RAM Usage

Figure 1: Box plots aggregated from execution of each element of the test set ($n = 616$).

proximity of the results for the `Column` and `Diagonal` methods, we calculated their 95 % confidence intervals. These intervals are $[5.69, 5.77]$ for the `Column` method and $[5.50, 5.58]$ for the diagonal method. The lack of overlap clearly indicates a statistically significant ordering, which is further confirmed by a pairwise t-test yielding a p-value approaching 0.

The reasons for these memory variations are also twofold. First, the total number of keys directly impacts memory consumption. The `Element`-wise method requires the most keys and consequently exhibits the highest usage. Interestingly, although the `Row` and `Column` methods both require only 6 keys, the latter requires 1.25 times more memory. This discrepancy again lies in the algorithmic design. The `Column` method must replicate the elements into separate ciphertexts; storing these additional elements increases memory consumption. It should be noted that while such minor algorithmic differences may seem negligible in the plaintext domain, this is not the case in the encrypted domain. To achieve the homomorphic property, very large numbers and complex algebraic structures are necessary, all of which consume substantial memory. Consequently, standard elements such as ciphertexts and evaluation keys can easily reach sizes of several hundred megabytes, or even gigabytes. Consequently, this clarifies why even a small network exhibits significant memory consumption.

Thus, we conclude that the `Row` method is the most efficient approach in terms of memory usage. The alternative methods require approximately 1.25 to 1.46 times more memory, though they remain viable options provided that memory capacity is not the primary system bottleneck.

6 Conclusion

In conclusion, there are significant differences between the methods in terms of execution time and memory usage. The `Diagonal` method is the fastest, whereas the `Row` method has the lowest memory footprint. Each method ranks second for the other metric. Therefore, choosing between them represents a practical trade-off. The difference in execution time is larger than that in memory usage; thus, if both constraints are important, the `Diagonal` method is preferable due to its strong overall performance. `Column` and `Element` consistently rank third and fourth, respectively. Consequently, we do not recommend their use, especially the gap between `Element` and the others is substantial.

Future work should further utilize HE-specific SIMD capabilities and expand the benchmark to larger networks and other architectures, such as Convolutional Neural Networks (CNNs).

References

[1] A. Tsanas and A. Xifara. “Accurate Quantitative Estimation of Energy Performance of Residential Buildings Using Statistical Machine Learning Tools”. In: *Energy and Buildings* 49 (June 2012), pp. 560–567.

[2] T. Prantl et al. “De Bello Homomorphico: Investigation of the Extensibility of the OpenFHE Library with Basic Mathematical Functions by Means of Common Approaches Using the Example of the CKKS Cryptosystem”. In: *International Journal of Information Security* 23.2 (Apr. 2024), pp. 1149–1169.

[3] S. Engel et al. “Homomorphic Approximation of Non-Linear Activation Functions Using the CKKS Cryptosystem of the OpenFHE Library”. In: *Discover AI* 6.1 (Aug. 2026), p. 891.

[4] L. J. Horn. *Laying the Foundation for Homomorphic Training of Neural Networks*. BestMasters. Springer Vieweg Wiesbaden, 2026.

[5] J. H. Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *ASIACRYPT 2017*. Ed. by T. Takagi and T. Peyrin. Springer International Publishing, pp. 409–437.

[6] S. Halevi and V. Shoup. “Algorithms in HElib”. In: *CRYPTO 2014*. Ed. by J. A. Garay and R. Gennaro. Springer, pp. 554–571.

[7] L. Horn et al. “Classification in the Dark: Performance Insights and Novel Methods for Homomorphic Binary Networks”. In: *WAHC 2026*.

Author's Version