A Tile Tensors Framework for Large Neural Networks on Encrypted Data

Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash,
Lev Greenberg, Ramy Masalha, Dov Murik, Hayim Shaul, Omri Soceanu

IBM Research

Abstract

Privacy-preserving solutions enable companies to offload confidential data to third-party services while fulfilling their government regulations. To accomplish that, they leverage various cryptographic techniques such as Homomorphic Encryption (HE), which allows performing computation on encrypted data. Most HE schemes work in a SIMD fashion, and the data packing method can dramatically affect the running time and memory costs. Finding a packing method that leads to an optimal performant implementation is a hard task.

We present a simple and intuitive framework that abstracts the packing decision for the user. We explain its underlying data structures and optimizer, and propose a novel algorithm for performing 2D convolution operations. We used this framework to implement an HE-friendly version of AlexNet, which runs in 3 minutes, more than 100 times faster than the state-of-the-art.

1 Introduction

Fully Homomorphic Encryption (FHE) schemes allow computations to be performed over encrypted data while providing data confidentiality for the input. Specifically, they allow the evaluation of functions on encrypted input, which is useful when outsourcing sensitive data to a third-party cloud environment. For example, a hospital that provides an X-ray classification service (e.g., COVID-19 versus pneumonia) can encrypt the images using FHE, express the classification algorithm as a function, and ask a cloud service to evaluate it over the encrypted data without decrypting it. In this way, the hospital can use the cloud service while still complying with regulations such as HIPAA [9] and GDPR [15].

The proliferation of FHE solutions in the last decade attempts to address users' security requirements while providing efficient solutions in terms of time and memory. Nevertheless, it turns out that running large NNs using FHE encryption is still considered an expensive task. For example, the best implementation of AlexNet [25], before this paper, takes 1.5 days. This barrier forces users to search for other secure alternatives instead of enjoying the advantages of solutions that use only FHE. Our proposed framework aims to narrow down this barrier for such FHE systems, allowing them to better utilize cloud capabilities while operating on their confidential data.

Some FHE schemes, such as CKKS [10], operate on ciphertexts in a homomorphic Single Instruction Multiple Data (SIMD) fashion. This means that a single ciphertext encrypts a fixed size vector, and the homomorphic operations on the ciphertext are performed slot-wise on the elements of the plaintext vector. To utilize the SIMD feature, we need to pack and encrypt more than one input element in every ciphertext. The packing choice can dramatically affect the *latency* (i.e., time to perform computation), *throughput* (i.e., number of computations performed in a unit of time), communication costs, and memory requirements. For example, we isolated the effect of different packing choices by testing them on CryptoNets [16]. We summarize the results in Table 4 and show that using two naïve packing solutions, we can achieve a latency of 0.86 sec. and 11.1 sec., with memory requirements of 1.58 GB and 14 GB, respectively. In comparison, a different non-trivial packing, achieves a latency of 0.56 sec. and memory requirements of 0.77 GB.

Accelerating different computations on encrypted data may require using different packing methods (e.g., as in [8, 19, 22, 24]). Consider, for example, the simple case of multiplying two $d \times d$ matrices. At one end of the spectrum, there is the simple matrix multiplication algorithm, for which every matrix is packed efficiently (concerning space) in a single ciphertext and requires $O(d^3)$ time. At the other end of the spectrum, [23] showed a packing method, which was later expanded by [2], that uses $O(d)$ times more space but takes only $O(d)$ time.

Deciding which packing to use is hard and the more efficient packing may not be the trivial one (see above). Even worse, different optimization goals may lead to different packing, e.g., as shown in Table 5. Moreover, as the evaluated circuit size increases, it becomes harder to find the optimal

packing. For example, finding the best packing for a large Neural Network (NN) inference algorithm is hard since it involves high dimensional data, where the input is typically a four or five dimensional tensor, and the computation involves long sequences of operations such as matrix multiplication and convolution.

When homomorphically evaluating a circuit, a common target for optimization is to reduce its multiplication depth. The reason is that every ciphertext operation, and specifically multiplication, adds some noise to the underlying plaintext. When the noise size crosses some precalculated limit, it becomes impossible to remove it and extract the data. Avoiding this situation can be done in two ways: a) using bootstrapping, which is a heavy ciphertext operation that "cleans" most of the noise from the underlying plaintext; b) asking the data owner for assistance, i.e., asking the user to decrypt the ciphertext, clean the noise from the plaintext, and re-encrypt the data using FHE. The latter option was implemented in GAZELLE [24] and NGraph [5] using Multi Party Computations (MPC). It has the drawback that the client must stay online during the computations or delegate its keys to a trusted third-party. On the other hand, client-aided solutions allow computing non-polynomial functions such as the ReLU activation function on the unencrypted plaintext.

Related work. Some recent FHE compilers [5, 14] simplify the way users can implement NN solutions on encrypted data by allowing them to focus on the network and leaving the packing optimizations to the compilers. This is also the purpose of our tile tensor framework. It enables us to evaluate an FHE-friendly version of AlexNet [25] in 3 minutes. To the best of our knowledge, this is the first time such a big network has been implemented with a feasible running time. In comparison, NGraph [5] reported their measurements for CryptoNets [16] or for MobileNetV2 [29] when using client-aided design, and CHET [14] reported the results for SqueezNet. All of these networks are smaller than AlexNet. Another experiment using NGraph and CHET was reported in [31] using Lenet-5 [26], which is also a small network compared to AlexNet. We note that we could not evaluate AlexNet on CHET because it was not freely available online at the time of writing this paper. We implemented AlexNet using NGraph but we terminated the experiment after 6 hours. TenSEAL [4] is another new library, where we were able to follow the tutorials and implement CryptoNets. However, we could not find a simple way to build a network with more than one convolution layer without considering packing, as required for AlexNet.

Our Contribution Our contributions can be summarized as follows:

- *A tile tensor based framework*. We introduce a new packing-free programming-framework that allows users to concentrate on the NN design instead of the packing decisions. This framework is simple and intuitive, and will be available for non-commercial use in the near future.

- *Packing optimizer*. We describe a packing optimizer that considers many different packing schemes. The optimizer estimates the time and memory needed to run the circuit with each scheme, and reports the one that optimizes a given objective, whether latency, throughput, or memory.

- *New 2D convolution-layer implementation using a novel packing.* We provide a new packing method and an implementation of 2D-convolution, which is a popular block in NNs. Our new packing and implementation are more efficient for large inputs than previous work. In addition, with this packing we are able to efficiently compute a long sequence of convolution-layers.

- *Efficient FHE-friendly version of AlexNet inference under encryption*. We implemented an FHE-friendly version of AlexNet. To the best of our knowledge, this is the fastest non-client-aided evaluation of this network.

The rest of the paper is organized as follows. Section 2 describes the notation used in the paper, and some background terminology. Section 3 provides an overview of the tile tensor framework and Section 4 introduces the tile tensors data structure. Section 5 describes the optimizer, Section 6 describes a novel convolution algorithm, and Section 7 shows experimental results for CryptoNets and AlexNet. In Section 8, we compare our methods with existing methods and we summarize our conclusions in Section 9.

2 Background

2.1 Notation

We use the term *tensor* as synonymous with multi-dimensional array, as this is common in AI. We denote the shape of a k-dimensional tensor by $[n_1, n_2, \ldots, n_k]$, where $0 < n_i$ is the size of the i'th dimension. For example, the shape of the 5×6 matrix M is $[5,6]$. We sometimes refer to a tensor M by its name and shape $M[5,6]$ or just by its name M when the context is clear. For a tensor R, we use $R(j_1, j_2, \ldots, j_k)$ to refer to a specific element, where $0 \leq j_i < n_i$. We use uppercase letters for tensors.

We write matrix multiplication without a multiplication symbol, e.g., M_1M_2 stands for the product of M_1 and M_2. We denote the transpose operation of a matrix M by M^T and we use tags (e.g., M', M'') to denote different objects.

2.2 Tensor Basic Operations

2.2.1 Broadcasting and Summation

The tensor functions "broadcasting" and "summation" that we define in this section allow us to easily describe some algebraic operations on tensors such as vector-matrix and matrix-matrix multiplication. We start by first defining the term *compatible shape* for tensors.

Definition 2.1 (Compatible shapes)**.** The tensors $A[n_1,\ldots,n_k]$ and $B[m_1,\ldots,m_k]$ have *compatible shapes* if $m_i = n_i$ or either $n_i = 1$ or $m_i = 1$, for $i < k$. Their *mutual expanded shape* is $[\max\{n_i,m_i\}]_{i<k}$.

Remark 1. When a tensor A has more dimensions than a tensor B, we can match their dimensions by expanding B with dimensions of size 1. This results in equivalent tensors up to transposition. For example, both tensors $V[b]$ and $V[b,1]$ represent column vectors, while $V[1,b] = V^T$ represents a row vector.

The broadcasting operation takes two tensors with compatible but different shapes and expands every one of them to their mutual expanded shape.

Definition 2.2 (Broadcasting)**.** For a tensor $A[n_1,\ldots,n_k]$ and a tensor shape $s = [m_1,\ldots,m_k]$ with $n_i \in \{1,m_i\}$ for each $i = 1,\ldots,k$. The operation $C = broadcast(A,s)$ replicates the content of A along the r dimension m_r times for every $r = 1,\ldots,k$ and $n_r = 1 < m_r$. The tensor C is of shape s.

Example 1. The tensors $A[3,4,1]$ and $B[1,4,5]$ have compatible shapes. Their mutual expanded shape is $s = [3,4,5]$ and $broadcast(A,s)$ has the same shape as $broadcast(B,s)$.

We perform element-wise operations such as addition $(A+B)$ and multiplication $(A*B)$ on two tensors with compatible shapes A,B by first using broadcasting to expand them to their mutual expanded shape and then performing the relevant element-wise operation. The broadcasting step is degenerated when A and B are of the same shape. Figure 1 illustrates element-wise addition for a matrix $M[5,4]$ and a row vector $V[1,4]$.

Definition 2.3 (Summation)**.** For a tensor $A[n_1,\ldots,n_k]$, the operation $S = sum(A,t)$ sums the elements of A along the tth dimension for $1 \le t \le k$, i.e., for $i = 1,\ldots,(t-1),(t+1),\ldots,k$ and $j_i < n_i$

$$S(j_1,\ldots,j_{t-1},1,\ldots,j_k) = \sum_{l=0}^{n_t-1} A(j_1,\ldots,j_{t-1},l,\ldots,j_k).$$

The shape of S is $[n_1,\ldots,n_{t-1},1,\ldots,n_k]$.

Using broadcasting and summation we can define common algebraic operators.

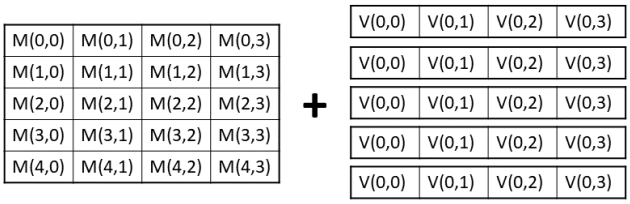

Figure 1: Element-wise addition of the matrix $M[5,4]$ and the vector $V[1,4]$. We first broadcast V using $broadcast(V,[5,4])$ so that its shape matches the shape of M. The illustrated addition operation can be interpreted as adding the row vector V to every row of M.

Example 2. For two matrices $M_1[a,b]$, $M_2[b,c]$ and the column vector $v[b,1]$, we compute matrix-vector multiplication using $M_1 v = sum(M_1 * v^T, 2)$ and matrix-matrix multiplication using $M_1 M_2 = sum(M_1' * M_2', 2)$, where $M_1' = M_1[a,b,1]$ and $M_2' = M_2[1,b,c]$.

2.2.2 Convolution

2D-convolution is a popular block in NNs. Its input is often an images tensor $I[w_I,h_I,c,b]$ and a filters tensor $F[w_F,h_F,c,f]$ with the following shape parameters: width w_I,w_F, height h_I,h_F, and the number of image channels c (e.g., 3 for an RGB image). In addition, we usually compute the convolution for a batch of b images and we compute the convolution results for f filters. Informally, the convolution operator moves each filter in F as a sliding window over every element of I where it can fit, and computes the inner product of each point in it with each corresponding point of I.

Definition 2.4 (Convolution)**.** Let $I[w_I,h_I,c,b]$ and $F[w_F,h_F,c,f]$ be two input tensors for the convolution operator representing images and filters, respectively. The results of the operation $O = convolution(I,F)$ is the tensor $O[w_O,h_O,f,b]$, where $w_O = w_I - w_F + 1$, $h_O = h_I - h_F + 1$, and

$$O(i,j,m,n) = \sum_{k=0}^{w_F-1}\sum_{l=0}^{h_F-1}\sum_{p=0}^{c-1} I(i+k,j+l,p,n)F(k,l,p,m). \tag{1}$$

In the degenerated case where $b = f = c = 1$, Equation (1) can be simplified to

$$O(i,j) = \sum_{k=0}^{w_F-1}\sum_{l=0}^{h_F-1} I(i+k,k+l)F(k,l). \tag{2}$$

2.3 Homomorphic Encryption

An FHE scheme is an encryption scheme that allows us to evaluate any circuit, and in particular any polynomial, on encrypted data. A survey is available in [18]. Common FHE instantiations include the following methods:

- $Gen(params)$ gets parameters *params* that depend on the scheme and generates the keys *pk* and *sk*.
- $Enc_{pk}(m)$ gets a message m and outputs a ciphertext c.
- $Dec_{sk}(c)$ gets a ciphertext c and outputs a message m'.
- $Add(c^a, c^b)$ gets two ciphertexts c^a, c^b and outputs a ciphertext c^{add}.
- $Mul(c^a, c^b)$ gets two ciphertexts c^a, c^b and outputs a ciphertext c^{mul}.
- $Rot(c^a, n)$ gets a ciphertext c^a and an integer n and outputs a ciphertext c^{rot}.

With the SIMD feature, the message is an s-dimensional vector $m = (m_1, \ldots, m_s)$, where s is the *slot count* and is determined by *params*. Similarly $Dec_{sk}(c) = m' = (m'_1, \ldots, m'_s)$. We denote $(Dec_{sk}(c))_i = m'_i$. An *exact* scheme such as [7] is correct if for each $i = 1, \ldots, s$ we have

$$\begin{aligned} m_i &= (Dec_{sk}(c))_i \\ (Dec_{sk}(Add(c^a, c^b)))_i &= (Dec_{sk}(c^a))_i + (Dec_{sk}(c^b))_i \\ (Dec_{sk}(Mul(c^a, c^b)))_i &= (Dec_{sk}(c^a))_i \cdot (Dec_{sk}(c^b))_i \\ (Dec_{sk}(Rot(c^a, n)))_i &= (Dec_{sk}(c^a))_{(i+n) \mod s} \end{aligned}$$

An *approximation* scheme, such as [10], is correct up to some small error term, i.e., $|m_i - Dec_{sk}(c^m)_i| \leq \varepsilon$, for some $\varepsilon > 0$ that is determined by *params*. For more details see [18].

3 Our Tile Tensor Framework

FHE libraries such as HElib and SEAL provide simple APIs for their users (e.g., encrypt, decrypt, add, multiply, and rotate). Still, writing an efficient program that involves more than a few operations is not always straightforward. As an example, consider the different methods for performing matrix-matrix multiplication that we mentioned in Section 1. Another example is the new convolution operator that we introduce in Section 6.4.

Providing users with the ability to develop complex and scalable FHE-based programs is the motivation that drives the ecosystem to develop higher-level solutions such as our library, NGraph [5], and CHET [14]. These solutions rely on the low-level FHE libraries while offering additional dedicated optimizations, such as accelerating NNs inference on encrypted data. Higher-level libraries optimize the user program at different abstraction layers. Figure 2 provides a simplified schematic view of the layers that we use in our library.

The first two layers include the low-level FHE libraries and their underlying SW/HW math accelerators. Every optimization for these libraries will automatically affect all the layers above them.

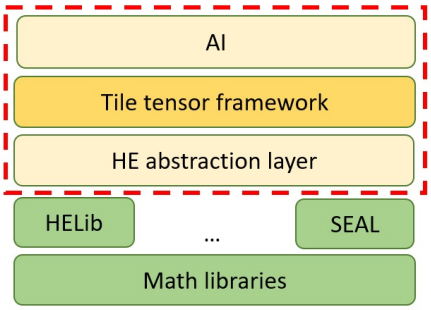

Figure 2: A simplified schematic illustration of the layers in our library.

Our library involves the three yellow upper layers. The bottom of these layers is the *HE abstraction layer* that makes our library agnostic to the underlying FHE library. The next layer is the *tile tensor framework layer*. It contains the tile tensor data structure (Section 4) that simplifies computation involving tensors, and the packing optimizer (Section 5) that searches for the most efficient packing configuration for a given computation. Together, they allow for a simple and efficient implementation of the AI layer above it.

In this paper we focus on the tile tensor framework layer, and specifically how it contributes to the optimization of NN inference computations.

4 Tile Tensors

In this section we informally introduce the tile tensor data structure [1].

4.1 Tiling Basics

We start by defining a simple tiling process in which we take a large tensor $A[n_1, n_2, \ldots, n_k]$, and break it up into smaller, equal-size blocks, which we call *tiles*, each having the shape $[t_1, t_2, \ldots, t_k]$.

For $i = 1, \ldots, k$, let $e_i = ceil(\frac{n_i}{t_i})$. We construct a tensor $E[e_1, e_2, \ldots, e_k]$, which we'll call the *external tensor*, such that each element of E is a tile. Thus, $T = E(a_1, a_2, \ldots, a_k)$ for $0 \leq a_i < e_i$ is a specific tile in E, and $T(b_1, b_2, \ldots, b_k)$ for $0 \leq b_i < t_i$ is a specific slot inside this tile. An element of the original tensor $A(c_1, c_2, \ldots, c_k)$ will be mapped to tile indices $a_i = \lfloor \frac{c_i}{t_i} \rfloor$, and indices inside the tile $b_i = c_i \mod t_i$. All other slots in E that were not mapped to any element of A will be set to 0.

For example, Figure 3a shows this tiling process applied to a matrix $M[5,6]$ and using tiles of shape $[2,4]$. The external tensor in this case has the shape $[3,2]$.

4.2 The Tile Tensor Data Structure

A tile tensor is a data structure containing an external tensor as described above, and meta data called *tile tensor shape*. The tile tensor shape defines the shape of the tiles, the shape of the original tensor we started with, and some additional packing details we describe later.

We use a special notation to denote tile tensor shapes. For example, $[\frac{n_1}{t_1}, \frac{n_2}{t_2}, \ldots, \frac{n_k}{t_k}]$ is a tile tensor shape specifying that we started with a tensor of shape $[n_1, \ldots, n_k]$ and tiled it using tiles of shape $[t_1, \ldots, t_k]$. In this notation, if $t_i = 1$, then it can be omitted. For example, $[\frac{5}{1}, \frac{6}{8}]$ can be written $[5, \frac{6}{8}]$.

A tile tensor can be created using a *pack* operation that receives a tensor A to be packed and the desired tile tensor shape: $T_A = pack(A, [\frac{n_1}{t_1}, \ldots, \frac{n_k}{t_k}])$. Since T_A contains both the external tensor created by the tiling process, and the tile tensor shape storing information about the original shape of A, we can retrieve A back using the *unpack* operation: $A = unpack(T_A)$. As with regular tensors, we sometimes refer to a tile tensor T_A together with its shape: $T_A[\frac{n_1}{t_1}, \ldots, \frac{n_k}{t_k}]$.

Figure 3 shows three examples of packing $M[5,6]$ into tile tensors with different tile tensor shapes.

4.3 Replication

A tile tensor shape can further indicate replication. If the i'th dimension in the tile tensor shape is $\frac{*}{t_i}$, then it implies $n_i = 1$, and during the packing process the tensor being packed is first broadcasted to have size t_i along this dimension. The unpacking process shrinks the tensor back to its original size. The replications can either be ignored, or an average of them can be taken (useful in case the data is stored in a noisy medium, as in approximate FHE schemes).

Figure 4 shows two ways to pack $V[5,1]$, with and without replication. In 4b, during the packing process we first compute $V' = broadcast(V, [5,4])$, then tile V' in the usual manner. The unpacking process will retrieve the original V.

4.4 Unknown Values

When tensors are packed into tile tensors, unused slots are filled with zeroes, as shown in Figures 3 and 4. However, after tile tensors are manipulated, the unused slots might get filled with arbitrary values, as explained in the next subsection. Although these unused slots are ignored when the tile tensor is unpacked, the presence of arbitrary values in them can still impact additional manipulation. To reflect this state, the tile tensor shape contains an additional flag per dimension, denoted by the symbol "?", indicating the presence of unknown values.

Figure 5 shows a tile tensor with the shape $[\frac{5}{2}, \frac{1?}{4}]$. The "?" in the second dimension indicates that whenever we exceed the valid range of the packed tensor along this dimension, we may encounter arbitrary unknown values. However, it still holds that $V = unpack(T_V)$, as these unused slots are ignored.

M(0,0)	M(0,1)	M(0,2)	M(0,3)	M(0,4)	M(0,5)	0	0
M(1,0)	M(1,1)	M(1,2)	M(1,3)	M(1,4)	M(1,4)	0	0
M(2,0)	M(2,1)	M(2,2)	M(2,3)	M(2,4)	M(2,5)	0	0
M(3,0)	M(3,1)	M(3,2)	M(3,3)	M(3,4)	M(3,5)	0	0
M(4,0)	M(4,1)	M(4,2)	M(4,3)	M(4,4)	M(4,5)	0	0
0	0	0	0	0	0	0	0

(a) $M[5,6]$ packed inside $T_M[\frac{5}{2}, \frac{6}{4}]$

M(0,0)	M(0,1)	M(0,2)	M(0,3)	M(0,4)	M(0,5)	0	0
M(1,0)	M(1,1)	M(1,2)	M(1,3)	M(1,4)	M(1,5)	0	0
M(2,0)	M(2,1)	M(2,2)	M(2,3)	M(2,4)	M(2,5)	0	0
M(3,0)	M(3,1)	M(3,2)	M(3,3)	M(3,4)	M(3,5)	0	0
M(4,0)	M(4,1)	M(4,2)	M(4,3)	M(4,4)	M(4,5)	0	0

(b) $M[5,6]$ packed inside $T'_M[5, \frac{6}{8}]$

M(0,0)	M(0,1)	M(0,2)	M(0,3)	M(0,4)	M(0,5)
M(1,0)	M(1,1)	M(1,2)	M(1,3)	M(1,4)	M(1,5)
M(2,0)	M(2,1)	M(2,2)	M(2,3)	M(2,4)	M(2,5)
M(3,0)	M(3,1)	M(3,2)	M(3,3)	M(3,4)	M(3,5)
M(4,0)	M(4,1)	M(4,2)	M(4,3)	M(4,4)	M(4,5)
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

(c) $M[5,6]$ packed inside $T''_M[\frac{5}{8}, 6]$

Figure 3: $M[5,6]$ packed into three different tile tensors with different tile tensor shapes. The rectangles represent the tiles. For each tile tensor, we show how M's elements are placed inside the tiles.

4.5 Interleaved Tiling

Another option for tiling is denoted by the symbol "$\sim$" in the tile tensor shape. This symbol indicates that the tiles do not cover a contiguous block of the tensor, but are spread out in equal strides. Using the notation of Subsection 4.1, an element of the original tensor $A(c_1, c_2, \ldots, c_k)$ will be mapped to tile indices $a_i = c_i \mod e_i$, and indices inside the tile $b_i = \lfloor \frac{c_i}{e_i} \rfloor$. See Figure 7a for an example.

For each dimension, we can specify separately whether it is interleaved or not. For example, in $[\frac{5}{2}, \frac{6\sim}{4}]$ only the second dimension is interleaved.

V(0,0)	0	0	0
V(1,0)	0	0	0
V(2,0)	0	0	0
V(3,0)	0	0	0
V(4,0)	0	0	0
0	0	0	0

(a) $V[5,1]$ packed inside $T_V[\frac{5}{2}, \frac{1}{4}]$

V(0,0)	V(0,0)	V(0,0)	V(0,0)
V(1,0)	V(1,0)	V(1,0)	V(1,0)
V(2,0)	V(2,0)	V(2,0)	V(2,0)
V(3,0)	V(3,0)	V(3,0)	V(3,0)
V(4,0)	V(4,0)	V(4,0)	V(4,0)
0	0	0	0

(b) $V[5,1]$ packed inside $T'_V[\frac{5}{2}, \frac{*}{4}]$

Figure 4: $V[5]$ packed into different tile tensors. The rectangles represents the tiles. For each tile tensor, we show how V's elements are placed inside the tiles.

V(0,0)	?	?	?
V(1,0)	?	?	?
V(2,0)	?	?	?
V(3,0)	?	?	?
V(4,0)	?	?	?
0	?	?	?

Figure 5: $V[5,1]$ packed in $T_V[\frac{5}{2}, \frac{1?}{4}]$. Unused space along the second dimension has unknown values, marked as cells containing ?.

Interleaved dimensions are useful for computing convolution, as explained in Section 6.

4.6 Tile Tensor Glossary and Notation

Below is a short summary of tile tensor terminology. Table 1 further summarizes the tile tensor notation options.

- *Tile tensor* A data structure containing an *external tensor* as data and a *tile tensor shape* as meta data.
- *External tensor* A tensor in which each element is a tile.
- *Packed tensor* The tensor that will be the result of unpacking a tile tensor.
- *Original shape* The shape of the packed tensor.

Notation	Meaning
$\frac{n_i}{t_i}$	Basic tiling
n_i	Basic tiling, $n_i = 1$
$\frac{*}{t_i}$	Replication, $n_i = 1$
$\frac{n_i?}{t_i}$	Unknown values
$\frac{n_i\sim}{t_i}$	Interleaved tiling

Table 1: Tile tensor shape notation summary

- *Tile shape* The shape of each tile in the external tensor.
- *Tile tensor shape* Meta data specifying the original shape, tile shape, and additional packing details.

4.7 Operators

Operators on tile tensors are defined by homomorphism with the packed tensors they contain. Let T_A and T_B be two tile tensors, and $\odot$ some binary operator, then $unpack(T_A \odot T_B) = unpack(T_A) \odot unpack(T_B)$. Unary operators are similarly defined.

Binary elementwise operators are implemented by applying the operation on the external tensors tile-wise, including broadcasting if needed. Similar to tensors, two tile tensors can be only be operated on if their shapes are compatible. Compatible tile tensor shapes have the same number of dimensions, and for each dimension specification they are either identical, or one is $\frac{*}{t_i}$ and the other is $\frac{n_i}{t_i}$. For example, $[\frac{18}{8}, \frac{4}{16}]$ is compatible with $[\frac{*}{8}, \frac{4}{16}]$. The intuition is that if the tensor is already broadcasted inside the tile, it can be further broadcasted to match any size by replicating the tile itself. In addition to computing the resulting external tensor, the resulting tile tensor shape should be computed as well, e.g., in some cases the replication is lost, and unknown values are introduced.

The *sum* operator is also defined homomorphically: $unpack(sum(T_A, i)) = sum(unpack(T_A), i)$. It works by summing over the external tensor along the i'th dimension, then by summing inside each tile along the i'th dimension. Assuming an FHE environment, this summation inside a tile requires a rotate-and-sum algorithm. The effect this has on the tile tensor shape who's i'th dimension is $\frac{n_i}{t_i}$ is as follows:

- If $t_i = 1$, then the resulting shape along the i'th dimension is $\frac{1}{1}$, or simply 1.

Operator	Resulting shape
$sum(T_A, 1)$	$[1, \frac{3}{8}, \frac{5}{16}]$
$sum(T_A, 2)$	$[4, \frac{*}{8}, \frac{5}{16}]$
$sum(T_A, 3)$	$[4, \frac{3}{8}, \frac{1?}{16}]$

Table 2: Summation rules example. The resulting shape after summing over any of the three dimensions of $T_A[4, \frac{3}{8}, \frac{5}{16}]$.

- If i is the lowest non-trivial tile dimension (i.e., the smallest i such that $t_i > 1$), the resulting shape along the i'th dimension is $\frac{*}{t_i}$.
- Otherwise, the resulting shape along the i'th dimension is $\frac{1?}{t_i}$.

If the dimension was $\frac{n_i?}{t_i}$ before summation, then after summation it will always be $\frac{1?}{t_i}$.

The reason for these rules lies in the rotate-and-sum algorithm. In a nutshell, the operator of rotating a tile can be used to rotate along dimensions, but for all dimensions except the first it becomes a shift operation in which elements falling of one side don't rotate back. On the first dimension, where we can actually rotate, summation ends up with replication.

As an example, let T_A be a tile tensor with the shape $[4, \frac{3}{8}, \frac{5}{16}]$. Table 2 depicts the resulting shape after summing over each of the three dimensions.

4.8 Higher Level Operators

Using elementwise operators and summation, we can perform various algebraic operations on tile tensors.

Matrix-vector multiplication. Given a matrix $M[a,b]$ and a vector $V[b]$, we reshape V to $V[1,b]$ for compatibility, and pack both tensors into tile tensors as $T_M[\frac{a}{t_1}, \frac{b}{t_2}]$, and $T_V[\frac{*}{t_1}, \frac{b}{t_2}]$, for some chosen tile shape $[t_1, t_2]$. We can multiply them using:

$$T_R[\frac{a}{t_1}, \frac{1?}{t_2}] = sum(T_M[\frac{a}{t_1}, \frac{b}{t_2}] * T_V[\frac{*}{t_1}, \frac{b}{t_2}], 2). \quad (3)$$

The above formula works for any value of a, b, t_1, t_2. This is because the tile tensor shapes of T_M and T_V are compatible, and therefore, due to the homomorphism, this computes $R[a,1] = sum(M[a,b] * V[1,b], 2)$, which produces the correct result as explained in Section 2.

A second option is to initially transpose both M and V and pack them in tile tensors $T_M[\frac{b}{t_1}, \frac{a}{t_2}]$ and $T_V[\frac{b}{t_1}, \frac{1}{t_2}]$. Now we can multiply them as:

$$T_R[\frac{*}{t_1}, \frac{a}{t_2}] = sum(T_M[\frac{b}{t_1}, \frac{a}{t_2}] * T_V[\frac{b}{t_1}, \frac{*}{t_2}], 2). \quad (4)$$

This computes the correct result using the same reasoning as before. The benefit here is that the result $T_R[\frac{*}{t_1}, \frac{a}{t_2}]$ is replicated along the first dimension due to the summation rules of Subsection 4.7. Thus, it is ready to play the role of T_V in Formula 3, and we can perform two matrix-vector multiplications consecutively without any processing in between.

Matrix-matrix multiplication. The above reasoning easily extends to matrix-matrix multiplication as follows. Given matrices $M_1[a,b]$ and $M_2[b,c]$, we can compute their product using either of the next two formulas, where in the second one we transpose M_1 prior to packing. As before, the result of the second fits as input to the first.

$$T_R[\frac{a}{t_1}, \frac{1?}{t_2}, \frac{c}{t_3}] = sum(T_{M_1}[\frac{a}{t_1}, \frac{b}{t_2}, \frac{*}{t_3}] * T_{M_2}[\frac{*}{t_1}, \frac{b}{t_2}, \frac{c}{t_3}], 2). \quad (5)$$

$$T_R[\frac{*}{t_1}, \frac{a}{t_2}, \frac{c}{t_3}] = sum(T_{M_1}[\frac{b}{t_1}, \frac{a}{t_2}, \frac{*}{t_3}] * T_{M_2}[\frac{b}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}], 1). \quad (6)$$

5 The Optimizer

The use of tile tensors in our library is transparent to the library users. In fact, to run a machine learning model inference, the users only need to supply the following inputs:

- The model architecture e.g., a NN architecture, and an indication whether the model weights will be encrypted.
- Requirements such as the inference batch size.
- Constraints such as the required security bits, precision, and limits on the maximal memory usage and computation time.
- Optimization targets such as CPU time or memory usage.

Internally, the packing optimizer chooses the most efficient packing arrangement for a given set of inputs while hiding this information from the user. In practice, there can be a large number of packing choices. For example, using an FHE scheme configured to have $16,384$ slots in each ciphertext, the tiles should be tensors with this many elements. Since our convolution operator (see Section 6) uses five-dimensional tiles, the number of possible shapes for them is $\binom{\log_2(16,384)+5-1}{5-1} = 3060$. The number of configurations is even higher when considering additional packing parameters.

Figure 6 presents a schematic illustration of the packing optimizer. The users provide a JSON file that contains the model architecture. The model unit processes this data and when the model architecture involves convolution layers, it identifies which convolution packing modes are supported (see Section 6.5), and passes this information to optimizer.

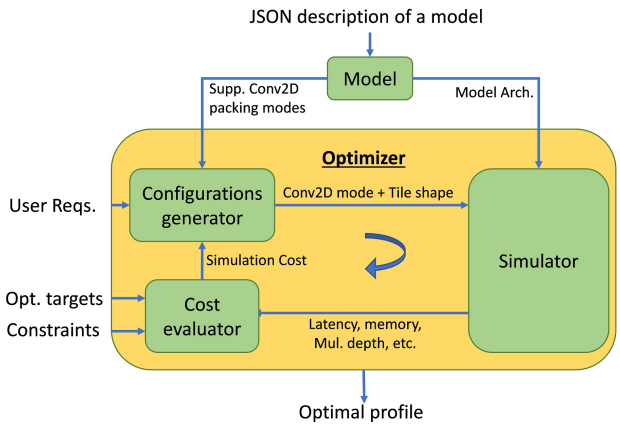

Figure 6: Packing optimizer

Packing optimizer. The packing optimizer involves three units: the configuration generator, the cost evaluator, and the simulator. The configuration generator generates a list of all possible packing configurations, including possible tile shapes and other options. The optimizer tests every configuration option using the simulator unit, which outputs the following data for every run: the computation time of the different stages including encrypting the model and input samples, running inference, and decrypting the results; the throughput; the memory usage of the encrypted model; input; and output; and more. The optimizer passes this data to the cost evaluator for evaluation. Finally, it returns the configuration option that yields the optimal cost to the user, together with the simulation output profile. A user can also use the optimizer to find the optimal configuration offline and cache the results for subsequent inference evaluations.

Cost evaluator. The cost evaluation unit computes the cost of running the model under a specific configuration option by evaluating the simulator output data, and considering the constraints and optimization targets provided by the user. In addition, the cost function involves the feasibility of the configuration with respect to the HE library. The final cost evaluation is sent to the configuration generator unit for optimization purposes.

Configuration generator. The configuration generator unit receives the model architecture and the list of the supported convolution packing methods, in case that a convolutional layer is present. It can use two strategies for generating configuration options. The first involves brute forcing over the possible configurations space, including all valid options for tile shapes. In the second strategy, for each convolution packing method, it will find the optimal tile shape using a "steepest ascent hill climbing" local search algorithm. It starts with a balanced tile shape, where the number of slots in every dimension is of the same order. This is a heuristic designed to avoid evaluating tile shapes that are likely to be computationaly costly at the beginning of the search. We then iteratively evaluate all the neighbor tile shapes of the current shape and continue to the best-improving neighbor as long as one exists. We consider two tile shapes as neighbors if we can obtain one shape from the other by multiplying or dividing the size of some of its dimensions by two. We consider one shape as better than another shape based on the costs received from the cost evaluator. Using the local search algorithm highly speeds up the search process and we found empirically that it often results in a global optimum. This was the case in our AlexNet and CryptoNets benchmarks.

Simulator. The simulator receives as inputs the model architecture from the model unit and a configuration option. At this stage, we can evaluate the configuration by running it on encrypted input under FHE. However, this will result in high resource consumption in terms of time and memory, even for a single configuration option, let alone thousands of configurations. Our simulator reduces these costs by taking a different approach. It uses pre-calculated benchmark values such as the CPU time of every HE operation and the memory consumption of a tile (i.e., the memory consumption of a single ciphertext). Then, it evaluates the model on mockup tile tensor objects using these benchmarks. Here, the tile tensors do not contain data and the tile tensor operations only accumulate resource information. Using this approach, the simulator can simulate an inference operation several order-of-magnitudes faster than when running the complete model on encrypted data. Section 7.3 reports the simulator accuracy on AlexNet.

5.1 Performance Evaluation

Table 3 demonstrates the advantage of combining the local search algorithm and the simulator when searching for an optimal configuration for AlexNet (see Section 11 for more details). Here, we fixed the number of slots to 16,384, the only feasible size for a NN that deep. We set the batch size to 1. The number of configuration options was 1360, with 680 different tile shapes for each convolution packing method. An exhaustive search that uses simulations took 13.8 minutes. In contrast, the local search algorithm took only 17 seconds and returned the same result. It did so after evaluating only 40 tile shapes.

The last column of Table 3 demonstrates the performance advantage of using the simulator over searching on FHE encrypted data. We heuristically estimated this data under the non-realistic assumption that memory is unlimited. To this end, we selected the four tile shapes with the lowest simulated time predictions when using local search ($ts = 1,\ldots,4$), and used them to measure the model runtime on three setups: a) using encrypted data while using only a single CPU thread e^1_{ts}; b) using encrypted data and 40 CPU

threads e^{40}_{ts}; c) using the simulations results for a single CPU thread s^1_{ts}. Subsequently, we estimated the advantage of using 40 threads by the ratio $threads\_ratio = avg(e^{40}_{ts}/e^1_{ts})$ and the simulation performance ratio by $sim\_ratio = avg(e^1_{ts}/s^1_{ts})$. Finally, we ran the simulator on all the feasible tile shapes and accumulated their normalized running times $\sum_t t \cdot sim\_ratio \cdot threads\_ratio$. The advantage of using the simulator is clear as it is $\times 2075$ and $\times 17000$ faster when performing local search and exhaustive search, respectively.

Search strategy	Evaluated states	Simulated search time	Estimated non-simulated search time
Exhaustive	1360	13.8 minutes	163 days
Local	40	17 seconds	9.8 hours

Table 3: A comparison of search strategies when using simulations versus running on encrypted data. In both cases, the reported data is for a setup with 40 CPU threads.

6 Convolution using Tile Tensors

In this section, we discuss how tile tensors can support the convolution operator. Although general, we describe it in the context of neural network inference since this is our leading motivation. In what follows, we assume the convolution problem in its simplest form: a single, one-channel image, $I[w_I, h_I]$, and a single filter $F[w_F, h_F]$. The output is $O[w_O, h_O]$. We extend it to multiple channels, filters, and batching in Subsection 6.4. In what follows we assume the slot count is s, i.e., each tile has s slots.

We first briefly describe in Subsection 6.1 and in Subsection 6.2 two naive packing methods. One is efficient in running-time and the other is efficient in storage. Then, in Subsection 6.3 and 6.4 we describe our novel approach, which is efficient in time and storage.

6.1 Time-efficient Naive Packing

We now describe a naive implementation of convolution using tile tensors that is efficient in running time. We pack I as $T_I[w_I, h_I, \frac{*}{s}]$, and the filter as $T_F[w_F, h_F, \frac{*}{s}]$. Recall that in our notation this means each tile has shape $[1,1,s]$, and each element of the input and filter matrices is mapped to a separate tile where its value is duplicated across all slots. This is inefficient with respect to storage but it allows for an easy implementation of Equation 2 homomorphically. Namely, we can substitute each element with its corresponding tile and perform the computation. The resulting tile tensor is $T_O[w_O, h_O, \frac{*}{s}]$.

This allows for a naïve utilization of the SIMD feature. If we have a batch of b images, given as a $I[w_I, h_I, b]$, we pack them as $T_I[w_I, h_I, \frac{b}{s}]$, i.e., with each image in a different slot. Computing a convolution with the same filter tile tensor $T_F[w_F, h_F, \frac{*}{s}]$ results with similarly packed output, $T_O[w_O, h_O, \frac{b}{s}]$.

Observation 2 (Single input)**.** Given an input image $I[w_I, h_I]$ and a filter $F[w_F, h_F]$, the packing described in Section 6.1 computes the convolution using: $O(w_I h_I w_F h_F)$ multiplications. The input is encoded in $O(w_I h_I)$ ciphertexts.

Observation 3 (Batched input)**.** Let s be the number of slots in a ciphertext. Then, given a batch of s images $I[w_I, h_I, s]$ and a filter tensor $F[w_F, h_F]$, with the packing described in Section 6.1, computing a convolution can be computed using (amortized cost) : $O(w_I h_I w_F h_F / s)$ multiplications. The input is encoded using $O(w_I h_I / s)$ ciphertexts (amortized).

6.2 Storage-efficient Naive Packing

We now describe a second naïve implementation. We pack I as $T_I[\frac{w_I}{t_1}, \frac{h_I}{t_2}]$, where $t_1 t_2 = s$. Recall this means I is divided into equal size tiles, each covering some contiguous block in I. (See Figure 3a for a concrete example.) The filter is packed as $T_F[w_F, h_F, \frac{*}{s}]$ as in Section 6.1. While the input is packed efficiently, computing Equation 2 homomorphically requires many rotations to correctly align the input.

6.3 Convolution with Interleaved Dimensions

We now show how interleaved dimensions (see Subsection 4.5) can be used to efficiently compute convolution.

Figure 7a shows a matrix $M[6,8]$ packed in the tile tensor $T_M[\frac{6\sim}{2}, \frac{8\sim}{4}]$. Here, the tiles' shape is $[2,4]$ and the external tensor shape is $[3,2]$. Every tile contains a 2×4 sub-matrix, but instead of being contiguous it is a set of elements spaced evenly in the matrix.

Figure 7b shows a different view of this packing scheme. For each element of the matrix, it shows the slot indices inside the tile to which it was mapped. For example, the top left block of 3×2 elements are all allocated to slot indices $(0,0)$. That is, each of these elements is placed in the $(0,0)$ slot of a different tile. Similarly, the next 3×2 block are mapped to the $(0,1)$ slot of a tile.

The interleaved packing allows for a more efficient implementation of Equation 2 with respect to runtime and storage. Intuitively, we use the SIMD to compute multiple element of the output in a single operation.

For example, Figure 8a shows a computation of the convolution output when the filter is placed at the top left position. The SIMD nature of the computation computes the output in other regions as well. The result is a single tile, where each slot contains the convolution result of the corresponding

M(0,0)	M(0,1)	M(0,2)	M(0,3)	M(0,4)	M(0,5)	M(0,6)	M(0,7)
M(1,0)	M(1,1)	M(1,2)	M(1,3)	M(1,4)	M(1,5)	M(1,6)	M(1,7)
M(2,0)	M(2,1)	M(2,2)	M(2,3)	M(2,4)	M(2,5)	M(2,6)	M(2,7)
M(3,0)	M(3,1)	M(3,2)	M(3,3)	M(3,4)	M(3,5)	M(3,6)	M(3,7)
M(4,0)	M(4,1)	M(4,2)	M(4,3)	M(4,4)	M(4,5)	M(4,6)	M(4,7)
M(5,0)	M(5,1)	M(5,2)	M(5,3)	M(5,4)	M(5,5)	M(5,6)	M(5,7)

Tile (0,0)

M(0,0)	M(0,2)	M(0,4)	M(0,6)
M(3,0)	M(3,2)	M(3,4)	M(3,6)

Tile (0,1)

M(0,1)	M(0,3)	M(0,5)	M(0,7)
M(3,1)	M(3,3)	M(3,5)	M(3,7)

Tile (1,0)

M(1,0)	M(1,2)	M(1,4)	M(1,6)
M(4,0)	M(4,2)	M(4,4)	M(4,6)

Tile (1,1)

M(1,1)	M(1,3)	M(1,5)	M(1,7)
M(4,1)	M(4,3)	M(4,5)	M(4,7)

Tile (2,0)

M(2,0)	M(2,2)	M(2,4)	M(2,6)
M(5,0)	M(5,2)	M(5,4)	M(5,6)

Tile (2,1)

M(2,1)	M(2,3)	M(2,5)	M(2,7)
M(5,1)	M(5,3)	M(5,5)	M(5,7)

(a) An $M[6,8]$ matrix packed in the tile tensor $T_M[\frac{6\sim}{2},\frac{8\sim}{4}]$.

S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)
S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)
S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)

Typical Tile

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)

(b) For each element of $M[6,8]$, the map show the slot indices in which it was placed.

Figure 7: Packing $M[6,8]$ into tile tensor $T_M[\frac{6\sim}{2},\frac{8\sim}{4}]$ filter-sized tiles, a packing method well-suited for convolution.

region, such that this tile is packed in the same interleaved packing scheme as the input tiles.

A more complicated example is given in Figure 8b. Here the filter is placed one pixel to the right. As a result, the filter needs to be multiplied by elements that appear in different regions, i.e. they are mapped to slots of different indices. In this case we need to rotate the tiles appropriately. For example, placing the filter with its upper left corner on pixel $(0,1)$, the convolution is computed using the $(0,0)$ slot of tiles $(0,1)$ and $(1,1)$ and slot $(0,1)$ of tiles $(0,0)$ and $(1,0)$. The latter two are therefore rotated to move the required value to slot $(0,0)$ as well.

The total cost of convolution when using this packing is summarized in the following lemma.

Lemma 4. Let s be the number of slots in a ciphertext. Then, given an input image $I[w_I,h_I]$ and a filter $F[w_F,h_F]$, packing I as $T_I[\frac{w_I\sim}{t_1},\frac{h_I\sim}{t_2}]$ and the filter as $T_F[w_F,h_F,\frac{*}{t_1},\frac{*}{t_2}]$, convolution can be computed using: $O(\lceil w_I h_I w_F h_F/s\rceil)$ multiplications, and $O(w_F\lceil\frac{w_I}{t_1}\rceil + h_F\lceil\frac{h_I}{t_2}\rceil + w_F h_F)$ rotations. The input is encoded in $O(w_I h_I/s)$ ciphertexts.

S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)
S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)
S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)

Tile (0,0)

M(0,0)	M(0,2)	M(0,4)	M(0,6)
M(3,0)	M(3,2)	M(3,4)	M(3,6)

Tile (0,1)

M(0,1)	M(0,3)	M(0,5)	M(0,7)
M(3,1)	M(3,3)	M(3,5)	M(3,7)

Tile (1,0)

M(1,0)	M(1,2)	M(1,4)	M(1,6)
M(4,0)	M(4,2)	M(4,4)	M(4,6)

Tile (1,1)

M(1,1)	M(1,3)	M(1,5)	M(1,7)
M(4,1)	M(4,3)	M(4,5)	M(4,7)

(a) Convolution with the filter at $M[0,0]$

S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)	...
S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)	...
S(0,0)	S(0,0)	S(0,1)	S(0,1)	S(0,2)	S(0,2)	S(0,3)	S(0,3)	...
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)	...
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)	...
S(1,0)	S(1,0)	S(1,1)	S(1,1)	S(1,2)	S(1,2)	S(1,3)	S(1,3)	

(b) Convolution with the filter at $M[0,1]$

Figure 8: Convolution of $M[6,8]$ when the filter is placed over specific locations

Proof. **Multiplications.** To compute the convolution we need to multiply each of the $w_I h_I$ elements of the input tensor with each of the $w_F h_F$ elements of the filter (excluding edge cases that do not change the asymptotic behavior). Since each multiplication multiplies s slots we need only $O(\lceil w_I h_I w_F h_F/s\rceil)$ multiplications.

Rotations. Recall the output is of size $(w_I - w_F + 1)(h_I - h_F + 1)$ where

$$O[x_o,y_o] = \sum_{i=0}^{w_F-1}\sum_{j=0}^{h_F-1} I[x_o+i,y_o+j].$$

We map to the k-th slot of different ciphertexts elements of I with indexes $k\lceil\frac{w_I}{t_1}\rceil \le x_o < (k+1)\lceil\frac{w_I}{t_1}\rceil$ and $k\lceil\frac{h_I}{t_2}\rceil \le y_o < (k+1)\lceil\frac{h_I}{t_2}\rceil$. It is therefore enough to analyze the cost of computing the convolution for $0 \le x_o < \lceil\frac{w_O}{t_1}\rceil$ and $0 \le y_o < \lceil\frac{h_O}{t_2}\rceil$, since computing the other elements of the output have no cost due to the SIMD feature.

It follows that a rotation is needed when $x_o + i \geq \lceil \frac{w_I}{t_1} \rceil$ or $y_o + j \geq \lceil \frac{h_I}{t_2} \rceil$. This totals to $O(w_F \lceil \frac{w_I}{t_1} \rceil + h_F \lceil \frac{h_I}{t_2} \rceil + w_F h_F)$.

Storage. Since we use $O(s)$ slots of each ciphertext, the input can be encoded in $O(w_I h_I / s)$ ciphertexts. □

6.4 Handling Multiple Channels and Filters

We now extend this result to handle channels, batch, and filters dimensions. We pack the tensor of images $I[w_I, h_I, c, b]$ as $T_I[\frac{w_I \sim}{t_1}, \frac{h_I \sim}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{*}{t_5}]$ and pack the filters $F[w_F, h_F, c, f]$, as $T_F[w_F, h_F, \frac{*}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}, \frac{*}{t_4}, \frac{f}{t_5}]$, where $t_i \in \mathcal{N}$ and $\prod t_i = s$.

The convolution is computed similarly to Section 6.3, multiplying tiles of T_I with the appropriate tiles of T_F. The result is a tile tensor of shape $T_O[\frac{w_O \sim ?}{t_1}, \frac{h_O \sim ?}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{f}{t_5}]$. Summing over the channel (the 3^{rd}) dimension, we obtain $T_O[\frac{w_O \sim ?}{t_1}, \frac{h_O \sim ?}{t_2}, \frac{1?}{t_3}, \frac{b}{t_4}, \frac{f}{t_5}]$.

6.5 A Sequence of Convolutions

In this section we discuss how to implement a sequence of multiple convolution layers. This is something that is frequent in neural network and involves some non-trivial details. One of the advantages of our tile tensor method is that the output of one convolution layer can be easily adjusted to be the input of the next convolution layer.

Assume we are given an input batch tensor, $I[w_I, h_I, c, b]$ and a sequence of convolution layers with the l'th layer having a filter tensor $F^l[w_F^l, h_F^l, c^l, f^l]$. For the first layer we have $c^1 = c$, and for $l > 1$ we have $c^l = F^{l-1}$.

As before, we pack the input tensor as $T_I[\frac{w_I \sim}{t_1}, \frac{h_I \sim}{t_2}, \frac{c}{t_3}, \frac{b}{t_4}, \frac{*}{t_5}]$.

For odd layers, $l = 2\ell + 1$, we pack the filter tensor as before $T_F^l[w_F^l, h_F^l, \frac{*}{t_1}, \frac{*}{t_2}, \frac{c}{t_3}, \frac{*}{t_4}, \frac{f^l}{t_5}]$. The output is then $T_O[\frac{w_O^l \sim ?}{t_1}, \frac{h_O^l \sim ?}{t_2}, \frac{1?}{t_3}, \frac{b}{t_4}, \frac{f^l}{t_5}]$.

For even layers, $l = 2\ell$, we introduce this packing for the filters: $T_F^l[w_F^l, h_F^l, \frac{*}{t_1}, \frac{*}{t_2}, \frac{f}{t_3}, \frac{*}{t_4}, \frac{c}{t_5}]$.

As can be seen, the shapes of layer outputs do not match the shapes of the inputs of the subsequent layers. We now show how to solve it and thus allow for a sequence of convolution layers.

To make an output of an odd layer suitable for the next even layer, we clear the unknowns by multiplying with a mask and then replicate the channel dimension. We then get a tile tensor of this shape: $T_O[\frac{w_O^l \sim ?}{t_1}, \frac{h_O^l \sim ?}{t_2}, \frac{*}{t_3}, \frac{b}{t_4}, \frac{f^l}{t_5}]$,

which matches the input format of the next layer since $f^l = c^{l+1}$. To make an output of an even layer suitable for the next odd layer, we similarly clean and replicate along the filter dimension.

We note that changing the order of the dimensions leads to a small improvement. The improvement comes because summing over the first dimension ends up with a replication over this dimension. Therefore, setting the channel dimension as first saves us the replication step when preparing the input to an even layer. Alternatively, the filter dimension can be set as first and then the replication step can be skipped when preparing the input to an odd layer.

7 Experimental Results

In this section we demonstrate our approach for neural networks inference under encryption.

Our method can work when either only the input to the network is encrypted, or only the network weights are encrypted, or both. In an FHE computation that involves both encrypted and non-encrypted data, the non-encrypted part undergoes an *encoding* step, which arranges it in objects containing the same number of slots as the ciphertexts, hence tile tensors are relevant for both types of data.

7.1 CryptoNets Benchmark

For this benchmark we use the CryptoNets network [16] described in Appendix A.1. The network was trained for classifying the MNIST dataset [27] and reaches an accuracy of 98.95%. This network starts with a convolutional layer followed by two fully connected layers.

Our method assumes a user specified batch size of n. In the tile tensor shape, the third dimension is reserved for batch size. By choosing the corresponding tile size along this dimension, t_3, we can efficiently adapt to a wide range of batch sizes.

Since the input images to this network are small, the convolutional layer was handled using a different approach than the one described in Section 6. For this network, we implemented a variant of the simple approach known as *image-to-column* [4]. For each filter F of the convolutional layer we identified all possible window locations on the input image. We extracted each such window, flattened it into a row, and created a matrix $M_1[845, 25]$ with all these rows replicated 5 times, once for each filter. A second matrix $M_2[845, 25]$ was populated with the corresponding flattened filter for each row. Computing the elementwise multiplication $M_1 * M_2$ and summing over the rows thus results in computing the convolution. We packed them as $T_{M_1}[\frac{25}{t_1}, \frac{845}{t_2}, \frac{n}{t_3}]$ and $T_{M_2}[\frac{25}{t_1}, \frac{845}{t_2}, \frac{*}{t_3}]$, computing the convolution as $sum(T_{M_1} * T_{M_2}, 1) = T_V[\frac{*}{t_1}, \frac{845}{t_2}, \frac{n}{t_3}]$. The result is thus flattened along the second dimension, and replicated along the first, making it ready as input for the first Fully Connected (FC) layer.

The first FC layer weights $W_1[100, 845]$ were packed as $T_{W_1}[\frac{100}{t_1}, \frac{845}{t_2}, \frac{*}{t_3}]$. Multiplying them with the input T_V, we obtain $sum(T_W * T_V, 2) = T_R[\frac{100}{t_1}, \frac{1?}{t_2}, \frac{n}{t_3}]$. The result's unknown values were cleaned by multiplying with a mask, and then replicated along the second dimension using rotate-and-sum, making it suitable as input for the second FC layer with weights $W_2[10, 100]$ packed as $T_{W_2}[\frac{100}{t_1}, \frac{10}{t_2}, \frac{n}{t_3}]$.

t_1	t_2	t_3	**Latency (sec)**	**Enc/Dec (sec)**	**Memory (GB)**
1	8192	1	0.86	0.04	1.58
8	1024	1	0.56	0.04	0.76
32	256	1	0.56	0.04	0.73
64	128	1	0.57	0.04	0.77
128	64	1	0.61	0.04	0.94
256	32	1	0.68	0.05	1.37
1024	8	1	1.93	0.14	3.17
8192	1	1	11.10	0.80	14.81

Table 4: The inference performance running CryptoNets with different tile sizes. We set $t_3 = n = 1$ and show some possible choices for t_1 and t_2. The first three columns show the tile shape $[t_1, t_2, t_3]$. Latency measures the time to complete an inference. The next column shows the time to encrypt the input and decrypt the output, and the last shows the RAM needed for the inference.

When setting $t_3 = s$ (where s is the number of slots in a ciphertext) and $t_1 = t_2 = 1$, the FC layers reduce to a known method sometimes referred to as SIMD representation (see Section 8). In this case, we computed the convolution in the more simple and efficient way described in Subsection 6.1.

Our experiments use CKKS [10], configured for 8192 slots. More technical details are given in Appendix B. Since the input's third dimension is $\frac{n}{t_3}$, the computations is most efficient when the batch size n equals t_3. This prevents unused slots and minimizes latency. Thus, our experiments assume $n = t_3$. For each $t_3 = 1, 2, 4, \ldots, 8192$, we tested all possible alternatives for t_1 and t_2. All the results are the average of 10 runs.

Table 4 summarizes some of the results for $t_3 = 1$. The most efficient tile shape is $[32, 256, 1]$, which achieves the optimum in all measures. The reason is that it allows storing the largest tensors in this computation, the two $[25,845]$ matrices (input and filters) and the first FC layer's $[100,845]$ matrix, with relatively few tiles, reducing both memory and CPU usage. These results indicate that for a batch size of 1, the optimizer will choose the $[32, 256, 1]$ for the tile sizes as this is superior in every way to the other alternatives.

Table 5 shows some of the results for $t_3 = 1, 16, 64, 256, 1024, 4096, 8192$ and batch size $n = t_3$. For each value of t_3, we show the optimal value of t_1, t_2. Here, each row is a reasonable choice since it offers a different tradeoff between the performance measures. When increasing t_3, the latency and memory consumption increase, but the per-sample amortized latency decreases. The encryption and decryption time also increase with t_3, except for $t_3 = 8192$. As mentioned above, for this case, we switched to the naïve way of computing convolution, which reduces some overhead in input packing.

7.2 AlexNet Benchmark

7.2.1 COVIDx classification over HE

COVIDx CT-2A Data-set An open access benchmark dataset designed by [17] was generated from several open datasets, and comprises 194,922 CT slices from 3,745 patients. It contains three classes of chest CT images: *Normal*, *Pneumonia* or *COVID-19* cases. For this experiment, we took a subset of 10K images per class for training, 1K images per class for validation, and 201 images in total for test with 67 random samples from each class. The size of the chosen test subset is small due to running time constraints.

Training a HE-friendly AlexNet model As a baseline, we used a variant of AlexNet network [25] that includes 5 convolution layers, 3 fully connected layers, 7 ReLU activations, 3 BatchNormalization layers, and 3 MaxPooling layers. The full network architecture appears in Appendix A.2. Following [3], we created a CKKS-compliant variant of AlexNet by replacing ReLU and MaxPooling components with a scaled square activation and AveragePooling correspondingly along with some additional changes; see more details in Appendix A.2. This model is trained on the COVIDx-CT training data set.

Prepare model for inference over encrypted data Since batch normalization requires division that is not a CKKS primitive, for inference we used a technique similar to [21] to "absorb" batch normalization layers into neighboring layers. This was done by modifying the neighbor layer's parameters in such a way that the resulting transformation of the layer is equivalent to a sequential application of batch normalization and the original layer. The resulting network is a computationally equivalent network, but doesn't include batch normalization layers. Similarly, we replaced the previously mentioned scaled square activation with x^2. In both cases, this approach helps reduce the multiplication depth of the network.

Another limitation when running under FHE is that numbers that grow too large during the computation may increase the noise, or exceed the allowed boundaries of the FHE scheme. These boundaries are most limiting in the last layers of the NN model. Therefore, we modified the network weights to avoid extremely large values while preserving network functionality.

Packing methods For the convolutional layers, we used the packing methods described in Subsection 6.5. The biases were similary packed in 5-dimensional tile tensors with compatible shapes, allowing us to add them to the convolution outputs.

The fully connected layers were handled using the matrix-matrix multiplication technique described in Subsection 4.8.

t_1	t_2	t_3	**Latency (sec)**	**Amortized Latency (sec)**	**Enc/Dec (sec)**	**Memory (GB)**
32	256	1	0.56	0.56	0.04	0.73
16	128	4	0.56	0.14	0.05	1.20
8	64	16	0.6	0.037	0.10	2.49
4	32	64	0.95	0.015	0.24	6.62
1	32	256	1.94	0.008	0.70	16.38
1	8	1024	5.6	0.0055	2.68	61.45
1	2	4096	21.57	0.0053	12.55	242.46
1	1	8192	41.32	0.005	1.29	354.47

Table 5: The inference performance running CryptoNets with different tile sizes. We show results for a range of t_3 values and for each value the optimal choice for t_1 and t_2 is shown. We set batch size $n = t_3$. The first three columns show the tile shape $[t_1, t_2, t_3]$. Latency measures the time to complete an inference, and amortized latency is the time divided by the batch size. The next column shows the time to encrypt the input and decrypt the output, and the last shows the RAM needed for the inference.

Since these are only three-dimensional, the first fully connected layer was packed as five-dimensional by artificially splitting its first dimension. We trimmed the extra two dimensions by combining three replicated dimension in its output to one.

More technical details are given in Appendix B.

7.2.2 AlexNet Benchmark Results

We evaluated our method's accuracy in the following environments. First, we used *vanilla AlexNet* executed in PyTorch [1] with test-set in plaintext. In the second environment, we had *HE-friendly AlexNet* executed in PyTorch with a test-set in plaintext. The third environment used *HE-AlexNet* executed in our framework with encrypted test-set. Table 6 shows the accuracy for each case. Transforming the vanilla AlexNet to be HE friendly reduces the accuracy by ~0.06. There is no additional degradation when running the HE-friendly model in our framework over an encrypted test-set.

When running under encryption, we compared the noise levels, runtime performance, and memory consumption on a set of 30 representative samples, with 4 different configurations. The first is *Plaintext-Latency*, which is optimized for low latency and the model's weights are in plaintext. The second configuration, *Plaintext-Throughput*, is optimized for high throughput and the model's weights are in plaintext. The third, *Ciphertext-Latency*, is optimized for low latency and the model's weights are encrypted. The fourth, *Ciphertext-Throughput*, is optimized for high throughput and the model's weights are encrypted. In all these configurations the input to the network is encrypted. We measure noise by comparing the result of the encrypted inference with the inference over an *HE-friendly AlexNet* in PyTorch, and calculating the root-mean-square-error (RMSE), maximum absolute error, and maximum relative error. The results are summarized in Table 7.

[1] PyTorch library https://pytorch.org

Environment	Accuracy
Vanilla AlexNet	0.861
HE-friendly AlexNet	0.806
HE AlexNet	0.806

Table 6: AlexNet accuracy evaluation in environments described in Section 7.2.2. Accuracy was measured on the test-set.

7.3 Optimizer Accuracy

Table 8 describes the results of an experiment that demonstrates the accuracy of the simulation mechanism. An inference over encrypted AlexNet model was performed using four different tile shapes, each with both simulated and actual encrypted computation. These shapes were chosen for being the four with lowest estimated predict time when searching using the local search strategy described in Section 5. The table includes the actual time an encrypted computation took and the deviation of the simulation estimation from it, for three relevant inference stages. The results show that the simulation mechanism provides relatively accurate time estimations for all four shapes. The estimated time deviated from the actual time by an average of -15.8%, -11.9% and -7.2% for predict, encryption of the model and encryption of a batch of input samples, respectively. Notice that the simulation mechanism provides time estimations assuming the computation is done on a single thread, and the comparison is also against non-simulated inference on a single thread. The simulation also

Configuration	Latency (sec)	Amortized Latency (sec)	Enc+Dec (sec)	Memory (GB)	RMSE	Max Absolute Error	Max Relative Error
Plaintext-Latency	181.9	181.9	5.3	123.8	1.72e−3	0.99e−2	1.78e−4
Plaintext-Throughput	720.8	90.1	5.4	568.1	1.75e−3	1.02e−2	1.75e−4
Ciphertext-Latency	358.1	358.1	5.4	223.4	2.00e−3	1.07e−2	1.67e−3
Ciphertext-Throughput	1130.4	282.6	5.6	688.8	3.57e−3	2.36e−2	6.18e−3

Table 7: AlexNet executed in our framework with different configuration. See configuration description in section 7.2.2

Tile shape	Packing mode	Inference time	Model encryption time	Input encryption time
$[16,8,8,16,1]$	CWHFB	4232 (-11%)	1509 (-11.5%)	162 (-6.8%)
$[8,8,8,32,1]$	CWHFB	4758 (-13.9%)	1493 (-12.1%)	164 (-7.9%)
$[16,8,8,16,1]$	FWHCB	4927 (-18.1%)	1680 (-11.5%)	177 (-6.8%)
$[32,8,8,8,1]$	FWHCB	4798 (-20%)	1668 (-12.3%)	178 (-7.3%)

Table 8: Accuracy of the simulated time estimations. All values are in seconds, the deviation of the estimated times from the real times are reported in brackets.

provides the expected storage taken by the encrypted model, encrypted input and output and the HE library context, which are not presented as they fully match the actual measures.

8 Comparison with State-of-the-Art

8.1 Matrix Multiplication

Multiple techniques for performing matrix multiplication under encryption have been presented, both as stand-alone methods, and as part of a larger framework, e.g., for NN inference.

A simple method is to pack each element of the input matrices in a separate ciphertext. This allows a straightforward implementation of matrix multiplication or any algorithm. Instead of using just one slot in each ciphertext, we can employ the additional ones for batching. This method is simple and has high throughput since there is no need for rotation operations. As a result, it is widely used under different names, "packing across the batch dimension", "packing the same dimension of multiple input samples in the same ciphertext", or "SIMD representation" [6, 8, 16, 28].

Tile tensors capture this approach as a special case. It can obtained for example by packing a batch b of matrices $M[x,y,b]$ as $T_M[x,y,\frac{b}{s}]$. However, if ciphertexts are large, it forces us to work in large batches, which may be memory intensive and not always practical. Also, if latency is the target measure for optimization and not throughput, this method is inefficient.

Crocket [12] shows a more sophisticated approach for matrix-vector multiplication. The vector is similarly divided into 2D blocks of the same size. The vector is laid out along one dimension and duplicated along the other. This method is more memory and time efficient for a single matrix-vector pair. It also allows for efficient consecutive applications of matrix-vector multiplications. The authors show an extension to matrix-matrix multiplication, by extracting columns from the second matrix and applying matrix-vector multiplication with each. The extraction of columns require increasing the multiplication depth and additional rotations.

Tile tensors capture this matrix-vector method as a special case, and generalize it in three respects. First, the original method has two separate algorithms: one for row-vector/matrix multiplication and one for matrix/column-vector multiplication. In tile tensors a single generalized algorithm handles both. Second, it allows adding a batch dimension as well, so the user can select its size. This offers a trade-off between latency and throughput, and a method to control memory usage, as will be demonstrated in our experimental results. Lastly, it naturally extends to matrix-matrix multiplication, without requiring additional rotations or increasing the multiplication depth.

The CHET compiler [14] uses a data structure termed CipherTensor. Like tile tensors, CipherTensor supports several packing techniques, and handles matrix-vector multiplication using a mix of multiplication and rotations. We believe CipherTensor is more rigid. It includes a fixed small set of implemented layouts, each with its own kernel of algorithms, whereas tile tensors offer a wider variety of options with a single set of generalized algorithms. Further, it wasn't demonstrated that CipherTensors offer an easy method to trade latency for throughput, and control memory consumption, as is possible in tile tensors by controlling the batch dimension.

Finally, CipherTensors require replicating the data of the input using rotations, whereas using tile tensors some of these replications can be avoided.

A different family of techniques are based on diagonalization. The basic method for matrix-vector multiplication is described in [19]. For a ciphertext with n slots, an $n \times n$ matrix is preprocessed to form a new matrix where each row is a diagonal of the original matrix. Then, multiplication with a vector can be done using n rotations, multiplications, and additions. Our method can achieve better performance by choosing square tiles of a shape approximating $[\sqrt{n}, \sqrt{n}]$. This allows us to perform the multiplication with n multiplications and $\sqrt{n}log\sqrt{n}$ rotations.

Some improvements to diagonalization techniques have been presented [11, 20]; these reduce the number of required rotations to $O(\sqrt{n})$ under some conditions, and by exploiting specific properties of the HE schemes of HElib [19]. Our methods make no special assumptions, but similarly exploiting such properties and combining them with the tile tensor data structure is reserved for future work.

In [22] a matrix-matrix multiplication method based on diagonalization is described. They reduce the number of rotations to $O(n)$ (instead of $O(n^2)$ for multiplying with n vectors). However, this comes at the cost of increasing the multiplication depth by 2 multiplications with plaintexts. Multiplication depth is usually the most expensive resource in an HE computation. The overall performance of the circuit is generally quadratic in depth, and from practical considerations the depth is sometimes bounded. Thus, an added 200% overhead in multiplication depth would severely harm the ability to perform deep computations such as inference over deep neural networks.

8.2 Convolution

A convolution layer is a basic building block in NN and previous work ([24] and [32]) addressed the problem of optimizing the implementation of convolution layers. In what follows, we discuss the previous implementations of convolution and compare them to our implementation.

Image Size. Previous work optimized for small input: Gazelle [24] considered a 28×28 grey scale images and GALA [32] considered 16×16 images. In our experiments we considered 224×224 RGB images. The previous work is less efficient for such large images. In a nutshell, they packed an entire image in a single ciphertext. Their improvement comes from packing several channels of an input image in a single ciphertext. For example, GALA requires a total of $O(\frac{f+cw_Ih_I}{c_n})$ permutation operations, where f, c, w_I, h_I are parameters as we report and c_n is the number of channels that are packed in a single ciphertext. With 224×224 and $65,536$ slots we have $c_n = 1$. If we have less slots, their performance degrades further since a single channel needs to be split between several ciphertexts.

Sequence of Convolution Layers. Previous works reported results for optimizing a single convolution layer. While this is important, deep networks have long sequences of convolution networks of different sizes and with different filters. For example, AlexNet has eight consecutive layers of convolution of different sizes. Previous works, assumed a non FHE step, such as garbled circuits or MPC, after each layer. This step performed the activation function and also put the input for the next layer in the correct format. Using these packing methods, an FHE-only system results in a very expensive step formatting the output of one layer to match the input of the next layer. As explained in Section 6.5, using the packing we propose, the formatting of an output of one layer to match the input of the next is very efficient in FHE.

8.3 Neural Network Inference

The LoLa network [8] works based on a a mixture of methods, manually tailored for a given use case. Switching between different methods within a single inference computation requires a processing stage between layers, resulting in extra additions and rotations. On the CryptoNets architecture they achieve a latency of 2.2 seconds using 8 threads. Our lowest latency is 0.56 seconds. The LoLa network uses 150 ciphertext-ciphertext multiplications, 279 rotations, and 399 additions for a single prediction. (We deduced these numbers from LoLa's detailed description.) Our approach requires 32 multiplications, 89 rotations, and 113 additions. This is roughly a four-fold reduction and matches the observed latency results. This demonstrates the efficiency of the tile tensor combined with an automatic optimization approach.

The CHET compiler [14] can perform inference of encrypted data in a non-encrypted network. For this easier problem, they report 2.5 seconds latency on a similarly sized, though less accurate, MNIST neural network classifier using 16 threads. They use a similar approach of an abstract data structure, CipherTensor, combined with automatic optimizations. We believe tile tensors are more flexible, as argued in the previous subsections, resulting in better optimization.

The EVA [13] compiler, built on top of CHET, improves the performance on the same network to 0.6 seconds using 56 threads and various optimizations unrelated to packing, of a kind outside the scope of this paper. Our best result on the more accurate CryptoNets architecture, when the network is not encrypted, goes down to 0.48 seconds. A direct comparison with EVA is difficult here due to multiple optimizations in EVA (e.g., eliminating rescale operations to reduce the overall prime chain length).

9 Conclusions

We presented a framework that acts as middleware between FHE schemes and the high-level tensor manipulation required in AI. Specifically, we demonstrated how our tile tensor based

framework can be used to improve latency for small networks, and scale up to much larger networks.

References

[1] Ehud Aharoni, Allon Adir, Moran Baruch, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Dov Murik, and Omri Soceanu. Tile tensors: A versatile data structure with descriptive shapes for homomorphic encryption. *CoRR*, abs/2011.01805, 2020. URL: https://arxiv.org/abs/2011.01805, arXiv:2011.01805.

[2] Adi Akavia, Hayim Shaul, Mor Weiss, and Zohar Yakhini. Linear-regression on packed encrypted data in the two-server model. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 21–32. ACM, 2019.

[3] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Sim Jun Jie, Benjamin Hong Meng Tan, Xiao Nan, Aung Mi Mi Khin, and Vijay Ramaseshan Chandrasekhar. Towards the AlexNet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs. *IEEE Transactions on Emerging Topics in Computing*, 2021. doi:10.1109/tetc.2020.3014636.

[4] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. *arXiv*, 2021. URL: https://arxiv.org/abs/2104.03152.

[5] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, pages 45–56, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338469.3358944.

[6] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. Ngraph-he: A graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, page 3–13, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3310273.3323047.

[7] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public-Key Cryptography - PKC 2013*, volume 7778, page 1, 2013. URL: https://www.iacr.org/archive/pkc2013/77780001/77780001.pdf, doi:10.1007/978-3-642-36362-7_1.

[8] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 812–821, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL: http://proceedings.mlr.press/v97/brutzkus19a.html.

[9] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at http://www.cms.hhs.gov/hipaa/, 1996.

[10] Jung Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of Advances in Cryptology - ASIACRYPT 2017*, pages 409–437. Springer Cham, 11 2017. doi:10.1007/978-3-319-70694-8_15.

[11] Jung Hee Cheon, Hyeongmin Choe, Donghwan Lee, and Yongha Son. Faster linear transformations in HElib , revisited. *IEEE Access*, 7:50595–50604, 2019. doi:10.1109/ACCESS.2019.2911300.

[12] Eric Crockett. A low-depth homomorphic circuit for logistic regression model training. Cryptology ePrint Archive, Report 2020/1483, 2020. https://eprint.iacr.org/2020/1483.

[13] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 546–561, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386023.

[14] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: An optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 142–156, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314628.

[15] EU General Data Protection Regulation. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural

persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, 119, 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016. URL: http://proceedings.mlr.press/v48/gilad-bachrach16.pdf.

[17] Hayden Gunraj, Ali Sabri, David Koff, and Alexander Wong. Covid-net ct-2: Enhanced deep neural networks for detection of covid-19 from chest ct images through bigger, more diverse learning. *arXiv preprint arXiv:2101.07433*, 2021. URL: https://arxiv.org/abs/2101.07433.

[18] Shai Halevi. Homomorphic Encryption. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 219–276. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-57048-8_5.

[19] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 554–571, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-44371-2_31.

[20] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. In *Annual International Cryptology Conference*, pages 93–120. Springer, 2018. doi:10.1007/978-3-319-96884-1_4.

[21] Alberto Ibarrondo and Melek Önen. Fhe-compatible batch normalization for privacy preserving deep learning. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 389–404. Springer, 2018. doi:10.1007/978-3-030-00305-0_27.

[22] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1209–1222, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243837.

[23] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1209–1222, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243837.

[24] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012. doi:10.1145/3065386.

[26] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

[27] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998. *URL http://yann. lecun. com/exdb/mnist*, 10:34, 1998.

[28] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. Towards deep neural network training on encrypted data. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 40–48, 2019. doi:10.1109/CVPRW.2019.00011.

[29] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. doi:10.1109/CVPR.2018.00474.

[30] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, April 2020. Microsoft Research, Redmond, WA.

[31] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully Homomorphic Encryption Compilers. *arXiv preprint arXiv:2101.07078*, pages 1–17, 2021. URL: http://arxiv.org/abs/2101.07078, arXiv:2101.07078.

[32] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. Gala: Greedy computation for linear algebra in privacy-preserved neural networks. *arXiv preprint arXiv:2105.01827*, 2021.

A Neural Networks Architecture

A.1 CryptoNets

Architecture defined in [16] with activation function $Act(x) = x^2$.

1. Conv2d: [Input: 28×28, 5 filters of size 5×5, stride=2, output: 845]+Act..
2. FC: [Input: 845, Output: 100]+Act.
3. FC: [Input: 100, Output: 10].

A.2 AlexNet

In this work we use the following variant of AlexNet network [25] as a baseline.

1. Conv2d(3, 64, kernel=11*11, stride=4, padding='same', activation= ReLU)
2. MaxPool2d(kernel=3*3, stride=2)
3. BatchNorm2d(64)
4. Conv2d(64, 192, kernel=5*5, stride=1, padding='same', activation=ReLU)
5. MaxPool2d(kernel=3*3, stride=2)
6. BatchNorm2d(192)
7. Conv2d(192, 384, kernel=3*3, stride=1, padding='same', activation=ReLU)
8. Conv2d(384, 256, kernel=3*3, stride=1, padding='same', activation=ReLU)
9. Conv2d(256, 256, kernel=3*3, stride=4, padding='same', activation=ReLU)
10. MaxPool2d(kernel=3*3, stride=2)
11. BatchNorm2d(256)
12. Dropout(p=0.2)
13. FC(in=9216, out=4096, activation=ReLU)
14. Dropout(p=0.2)
15. FC(in=4096, out=4096, activation=ReLU)
16. FC(in=4096, out=3)

In order to transform the model into a ckks-compatible model, three modifications made to the baseline architecture:

1. replace ReLU activation with a scaled square activation of the form $scaled\_square(x) = 0.01x^2$
2. replace MaxPooling with AveragePooling
3. replace the "same" padding with "valid" padding mode

While the first two modifications are necessary for a CKKS-compliant network, the third modification is required because of limitations of the current implementation of the tile tensors, that currently does not support padding

The resulting network is as follows:

1. Conv2d(3, 64, kernel=11*11, stride=4, padding='valid', activation= scaled_square)
2. AvgPool2d(3*3, stride=2)
3. BatchNorm2d(64)
4. Conv2d(64, 192, kernel=5*5, stride=1, padding='valid', activation=scaled_square)
5. AvgPool2d(kernel=3*3, stride=2)
6. BatchNorm2d(192)
7. Conv2d(192, 384, kernel=3*3, stride=1, padding='valid', activation=scaled_square)
8. Conv2d(384, 256, kernel=3*3, stride=1, padding='valid', activation=scaled_square)
9. Conv2d(256, 256, kernel=3*3, stride=1, padding='valid', activation=scaled_square)
10. AvgPool2d(kernel=3*3, stride=2)
11. BatchNorm2d(256)
12. Dropout(p=0.2)
13. FC(in=9216, out=4096, activation=scaled_square)
14. Dropout(p=0.2)
15. FC(in=4096, out=4096, activation=scaled_square)
16. FC(in=4096, out=3)

B Experiment results specifications

All experiments results reported in this paper use the same machine, an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz machine with 44 cores (88 threads) and 750GB memory. Unless explicitly specified otherwise, the experiments used only 40 threads and avoided hyperthreading by instructing the OpenMP library to pin one software thread per core.

We used the CKKS implementation in SEAL [30].

In the CryptoNets benchmark experiment, we used poly-degree 16384. The modulus chain was $\{45,35,35,35,35,35,45\}$ when either $t_1 = 1$ or $t_2 = 1$, and when both $t_1 > 1$ and $t_2 > 1$ the modulus chain was $\{45,35,35,35,35,35,35,45\}$, allowing a multiplication depth larger by 1, needed for replicating the results after layer 2 as is required by our alternating scheme. All results are the average of 10 runs.

In the AlexNet benchmark experiment, we used poly-degree 32768. The modulus chain was $\{53,43x18,53\}$, where $43x18$ stands for 18 values of size 43 each. All results are the average of at least 10 runs.