TENSEAL: A LIBRARY FOR ENCRYPTED TENSOR OPERATIONS USING HOMOMORPHIC ENCRYPTION

Ayoub Benaissa
École Supérieure en Informatique, Sidi Bel Abbès / OpenMined
Algeria
a.benaissa@esi-sba.dz

Bilal Retiat
École Supérieure en Informatique, Sidi Bel Abbès / OpenMined
Algeria
b.retiat@esi-sba.dz

Bogdan Cebere
OpenMined
bogdan.cebere@gmail.com

Alaa Eddine Belfedhal
École Supérieure en Informatique, Sidi Bel Abbès
Algeria
a.belfedhal@esi-sba.dz

ABSTRACT

Machine learning algorithms have achieved remarkable results and are widely applied in a variety of domains. These algorithms often rely on sensitive and private data such as medical and financial records. Therefore, it is vital to draw further attention regarding privacy threats and corresponding defensive techniques applied to machine learning models. In this paper, we present TenSEAL, an open-source library for Privacy-Preserving Machine Learning using Homomorphic Encryption that can be easily integrated within popular machine learning frameworks. We benchmark our implementation using MNIST and show that an encrypted convolutional neural network can be evaluated in less than a second, using less than half a megabyte of communication.

1 INTRODUCTION

In recent years, we have witnessed the evolution of machine learning as a service (MLaaS). In a typical scenario, the users need to send their input to the service provider, which will execute some algorithms on the data and send back the result.

This new way of making inferences has two critical issues. First, the users may not want to send their data to the service provider due to privacy concerns. Second, if we do not send users' data to the service provider, we cannot give the users the model due to intellectual property concerns. Using homomorphic encryption, we can follow the same method, except that users' data will always be encrypted. This way, neither the input nor the output will be visible to the service provider, and the evaluation can still happen on this encrypted data.

However, the adoption of homomorphic encryption in machine learning is slow. One reason is that while the available libraries provide an excellent API for cryptographers, they might be challenging to use for data scientists. The other blocker is also the cost for evaluation, both in terms of communication and computation.

1.1 CONTRIBUTIONS

- We present a flexible open-source library for doing encrypted tensor computation using homomorphic encryption. The library can directly convert tensors from popular machine learning frameworks (like PyTorch or Tensorflow) to their encrypted versions.
- We evaluate a convolution neural network on encrypted data in less than a second, with less than half a megabyte of communication during inference.

For the rest of the paper, we describe the library's architecture in Section 2. Then, in Section 3, we detail the algorithms needed for evaluating a convolutional neural network in the encrypted space. In Section 5, we provide an experimental evaluation of our library, and conclude with some limitations of our work in section 6.

2 ARCHITECTURE

TenSEAL is a library that bridges classical machine learning frameworks to homomorphic encryption capabilities. It manages all the complexities of implementing tensor operations on encrypted data. TenSEAL relies on the implementation of the CKKS (Cheon et al. (2017)) scheme in Microsoft SEAL. The clients can work with plain or encrypted tensors using one of the supported frontend languages (C++ or Python). In a client-server scenario, the message exchange is done using Protocol buffers. The core API is built around three main components: the context, the plain tensors, and the encrypted tensors.

2.1 THE TENSEAL CONTEXT

The TenSEAL context is the central component of the library. It generates and stores the necessary keys required by an encrypted computation. The context generates the secret-key used for decryption, the public-key used for encryption, the Galois-keys used for rotation, and the relinearization-keys used for relinearization of ciphertexts. This same object will also handle the thread-pool, which controls how many jobs should be run in parallel when performing parallelizable operations. The context can also be configured to do automatic ciphertext relinearization and rescaling during computation.

2.2 THE PLAINTENSOR

The PlainTensor is a class that connects unencrypted tensors to the encrypted implementations. Figure 2 in the appendix describes the process of converting the tensors.

2.3 ENCRYPTED TENSORS

The EncryptedTensor interface offers an API that needs to be implemented by every tensor exposed by the library. The interface has a TenSEALContext object, necessary to make any homomorphic encryption operation. The derived classes expose different tensor flavors, such as:

- CKKSVector derives the EncryptedVector interface and can hold a vector of real values by encrypting them into a single ciphertext.
- CKKSTensor follows the same strategy as Boemer et al. (2019b) and holds an N-dimensional tensor of real values by encrypting them into N-dimensional tensor of ciphertexts. However, it can batch an axis along with the slots available in every ciphertext, thus requiring only an (N-1)-dimensional tensor of ciphertexts.

Figures 1 and 2 in the appendix describe how an encrypted tensor is constructed. From this point onward, we will focus on CKKSVector, as it will be the type used to evaluate the MNIST dataset.

3 METHOD

When building a tensor over a Homomorphic Encryption scheme, there are two significant concerns to tackle: 1.How to encode the tensor before encryption? and 2.What operations can be performed

when using a particular encoding? The batching feature of the CKKS scheme allows a $N \times N$ matrix to be encrypted into N ciphertexts, with each row or column as a ciphertext. Another possibility is to encrypt the whole tensor into a single ciphertext (Jiang et al. (2018)). Depending on how we put our plain tensor into ciphertexts, we can perform different operations with varying complexities. The goal is to use the minimum number of ciphertexts and have a maximum depth with a minimum runtime, thus optimizing memory and computation. Seeking this ideal goal, we found out that we can use a single ciphertext to encrypt an input image and evaluate it on a convolutional neural network. This requires a pre-processing step on the client-side to encode the image as a matrix, composed of convolution windows as rows, then flatten it as a vector via a vertical scan. In TenSEAL, all these functionalities are implemented around CKKSVector. A CKKSVector holds $N/2$ real values, where N is the polynomial modulus degree. We can perform element-wise operations with other encrypted or plain vectors (addition, subtraction, and multiplication). We have a method for computing the power of an encrypted vector (element-wise) that uses an optimal circuit, thus using a minimum multiplicative depth. Also, because we need a polynomial approximation for different activation functions, we built a method for evaluating a polynomial with the encrypted vector as a variable, making sure to use a minimal circuit. Apart from the element-wise operations, we also need matrix operations to perform machine learning tasks. We implemented a variant of the encrypted vector-plain matrix multiplication proposed by Halevi & Shoup (2014) that can use multiple threads to run faster. You can check Table 3 in the appendix for a list of supported operations by the library's encrypted tensors.

3.1 Dot Product

The library provides a similar algorithm for dot product as in Halevi & Shoup (2014), but supports vectors of a size that is not a power of two or does not fill all the slots of a ciphertext. This limitation in the previous method was due to the right rotation that expects the final element to be the first, which was not true with regard to the cases we addressed. Our method is limited to a specific number of dot products if the vector size is not a power of two. However, this limitation is generally not reached, as the number of multiplication allowed by the scheme might be lower. We do this by replicating the input vector as many times as possible into ciphertext slots and only left ciphertext rotations during computation. Our method has the same algorithmic complexity as in Halevi & Shoup (2014). We implemented it using CKKS (Cheon et al. (2017)), for the dot product operation between an encrypted vector and a plain matrix. Thus, it can be extended to support a dot product between an encrypted matrix with a plain matrix (matrix multiplication). Figure 3 in the appendix shows how to perform a dot product between an encrypted vector and a plain matrix.

3.2 2-D Convolution

TenSEAL also supports evaluating convolutions, with a similar implementation of how modern machine learning frameworks (e.g., PyTorch) are computing them. We applied the Image Block to Columns (im2col) (Johnson et al. (2016)) technique, which turns a convolution layer into a single matrix multiplication operation. This technique requires encrypted matrix-plain vector multiplication, which we implemented by performing element-wise multiplication of the matrix transpose with replicated plain vector. Finally, it rotates and accumulates the result into a single vector. This operation uses only one multiplication operation and $log_2(N)$ rotations and additions, where N represents the rows' number in the matrix. Section A.4 in the appendix explains how the "image block to column" algorithm can be applied to encrypted inputs. It is important to note that the transformation happens in plain data, and the transformed input image will be encoded and encrypted in a single ciphertext. This directly implies that stacking two convolutions is not possible, as reorganizing the slots of a ciphertext is not trivial.

4 Related work

In recent years, several research works have made homomorphic encryption schemes practical for machine learning. Gilad-Bachrach et al. (2016) implemented CryptoNets, a neural network for making inference on encrypted data using the YASHE (Bos et al. (2013)) leveled homomorphic encryption scheme, which has efficient plain addition and multiplication algorithms, useful for unencrypted models. However, the framework requires large batches for achieving a good amortized

Framework	Method	Batch size	Message Size	Evaluation Time	Accuracy
CryptoNets	HE	8192	595.5 MB	570 s	99%
Gazelle	HE, MPC	1	0.5 MB	0.03 s	-
E2DM	HE	64	23.93 MB	1.69 s	98.1%
HCNN-GPU	HE	8192	-	5.16 s	99%

Table 1: Comparing frameworks and their evaluation results on MNIST.

performance, making it less practical for use cases that evaluate a single instance. Boemer et al. (2019b) used a similar tensor structure as CryptoNets while implementing different optimization layers. They used graph level optimizations specific for HE applications, which reduced the multiplicative depth needed for evaluating operations such as batch-norm and average pooling. In subsequent work, Boemer et al. (2019a) evaluated MobileNetV2 (Sandler et al. (2018)) and reported empirical results on HE encrypted inputs, which was way deeper than the previously used models. Jiang et al. (2018) used a smaller neural network and only convolution, linear layers, and the square activation function. Their framework E2DM made predictions on encrypted data using the CKKS (Cheon et al. (2017)) scheme, but compared to previous works, the model's parameters were also encrypted. Juvekar et al. (2018) proposed the Gazelle framework, which mixes HE with Garbled Circuits (GC) Yao (1986). They switched between both methods during computation, choosing the most efficient at a certain point based on the next operation. Using GC makes it possible to compute the ReLu activation function, compared to previous works (Gilad-Bachrach et al. (2016); Jiang et al. (2018); Boemer et al. (2019b;a)) which have mainly used polynomial functions. Even though the protocol achieves relatively fast run-time, it requires interaction between participants, resulting in high bandwidth usage. Badawi et al. (2020) used a GPU-accelerated implementation of BFV(Fan & Vercauteren (2012)) based on the work from Badawi et al. (2018). Their HCNN of 5 layers could evaluate in 5 seconds, but could batch more than 8000 images without extra overhead.

All the works are benchmarked using the MNIST dataset (LeCun et al. (2010)), but with different hardware configurations. We summarize empirical results reported in each of the corresponding papers in Table 1.

5 EVALUATION

To evaluate our library and technique, we implemented a neural network composed of: a convolutional layer (4 kernels of 7x7, with a stride of 3x3), a linear layer (input: 256, output: 64), and a final linear layer (input: 64, output: 10). We used the square activation function after every layer except for the last. The convolution was done using our image to column implementation, while the linear layers use the dot product implementation. The accuracy on the plain test-set was 97.7% in contrast to 97.4% for the encrypted test-set. We used the CKKSVector implementation, which uses the CKKS scheme. Knowing that we need 6 multiplications to perform the evaluation and a security level of 128-bits, we set the polynomial modulus degree to 8192, with a coefficient modulus of 206-bits, and a scale of 21-bits. The evaluation was done on Ubuntu Server 20.04 and Python 3.8, using AWS c4.2xlarge (8 vCPUs) and AWS c4.4xlarge (16 vCPUs) configurations. The measured durations are the average of 5 rounds of testing, with 10 iterations each. Table 2 contains a full breakdown for evaluating the neural network over encrypted images sampled from the MNIST dataset.

Operation	Description	Duration(ms)	
		AWS c4.2xlarge(8 vCPUs)	AWS c4.4xlarge(16 vCPUs)
Key generation	Generate the context and the encryption keys	940.01	921.04
Input preparation	im2col encoding	9.8	9.8
Convolutional layer evaluation	Input 28×28, kernel 7×7, stride 3, 4 channels	236.9	237.98
First activation(square)	Square 256 input values	8.47	8.42
FC1	Fully connected layer with 256 inputs and 64 outputs	1084.65	575.34
Second activation(square)	Square 64 input values	4.29	4.2
FC2	Fully connected layer with 64 inputs and 10 outputs	121.36	70.36
Full forward step	All the steps above	1456.29	887.06

Table 2: A complete illustration of the encrypted MNIST evaluation, with durations expressed in milliseconds. We evaluate the methods using two setups: Amazon c4.2xlarge instance (8 vCPUs, 15 GiB memory) and Amazon c4.4xlarge instance (16 vCPUs, 30 GiB memory), to underline how the library takes advantage of the available parallelism.

The results show that the library makes heavy use of the available parallelism, and it is highly competitive in terms of network communication, requiring only 427KB of communication to send the encrypted input and receive the encrypted output. At the same time, TenSEAL does not enforce a specific batch size for the inference, making it quite practical. The complete operations benchmarks are open-source, and the results are included in Section A.5 in the appendix. Table 4 shows the average performance for different arithmetic operations. Table 5 shows the average performance for matrix multiplications.

6 LIMITATIONS AND CONCLUSION

The encryption part relies on CKKS Cheon et al. (2017), which is known to be a leveled homomorphic encryption scheme. This means that depending on our parameter selection, there is a limit on how many multiplications we can perform on encrypted data, and this directly impacts the machine learning model we can use or its depth. Different machine learning models also use non-linear activation functions, which will need to be approximated using polynomials in the case of CKKS. A recent work Chillotti et al. (2020b) have been trying to solve this issue tightly related to machine learning by using the TFHE scheme Chillotti et al. (2020a), which allows the evaluation of deeper models, as well as non-linear activation functions.

In conclusion, our results show that it can be practical to do tensorial operations using the CKKS scheme. Depending on the use case, users can choose advanced tensor operations (like slicing or broadcasting) or use more computation-communication optimized implementations. TenSEAL can accommodate both scenarios while offering a smooth transition from the traditional machine learning frameworks. Finally, we seek to extend the tensor operations catalog and to improve the overall performance even further.

REFERENCES

Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Jun Jie Sim, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. In *IEEE Transactions on Emerging Topics in Computing*, 2020. doi: 10.1109/TETC.2020.3014636.

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*, pp. 45–56, 2019a.

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*, pp. 3–13, 2019b.

Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *IMA International Conference on Cryptography and Coding*, pp. 45–64. Springer, 2013.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 409–437. Springer, 2017.

Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020a.

Ilaria Chillotti, Marc Joye, and Pascal Paillier. New challenges for fully homomorphic encryption. *Privacy-preserving Machine Learning (PPML-PriML 2020) NeurIPS 2020 workshop*, 2020b.

Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.

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*, pp. 201–210, 2016.

Shai Halevi and Victor Shoup. Algorithms in helib. pp. 554–571, 2014.

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*, pp. 1209–1222, 2018.

Justin Johnson, Fei-Fei Li, and Andrej Karpathy. Cnns in practice. convolutional neural networks for visual recognitio. 2016.

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)*, pp. 1651–1669, 2018.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Microsoft SEAL. Microsoft SEAL (release 3.6). `https://github.com/Microsoft/SEAL`, November 2020. Microsoft Research, Redmond, WA.

Protocol buffers. Protocol buffers – Google's data interchange format. URL `https://github.com/protocolbuffers/protobuf`.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

TenSEAL. TenSEAL (release 0.3.0). `https://github.com/OpenMined/TenSEAL`, February 2021.

Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 162–167. IEEE, 1986.

A APPENDIX

A.1 ENCRYPTED TENSOR CLASSES

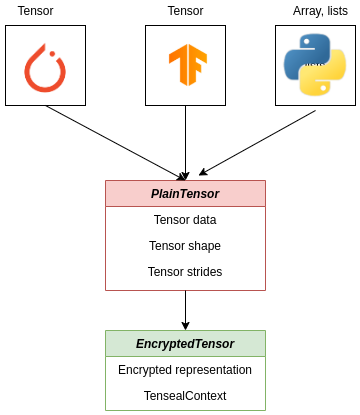

Figure 1: A high-level overview of the encrypted tensor construction. The PlainTensor wraps the tensor representation from popular frameworks, and it is used as input for the EncryptedTensor interface.

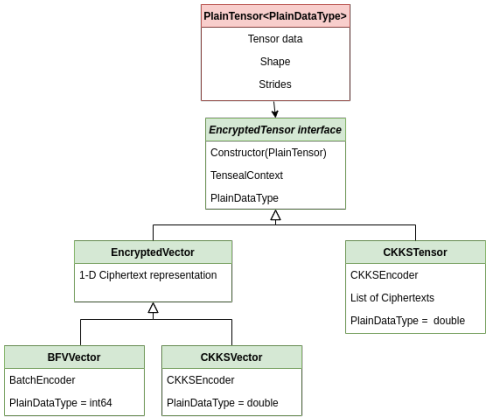

Figure 2: Encrypted tensors relation. The EncryptedTensor interface is derived into BFVVector, CKKVector, or CKKSTensor classes.

A.2 ENCRYPTED TENSOR OPERATIONS

Operation	Description
negate	Negate an encrypted tensor
square	Compute the square of an encrypted tensor
power	Compute the power of an encrypted tensor
add	Addition between an encrypted tensor and an encrypted/plain tensor
sub	Subtraction between an encrypted tensor and an encrypted/plain tensor
mul	Multiplication between an encrypted tensor and an encrypted/plain tensor
dot_product	Dot product between an encrypted tensor and an encrypted/plain tensor
polyval	Polynomial evaluation with an encrypted tensor as variable
matmul_plain	Multiplication between an encrypted tensor and an encrypted/plain matrix
conv2d_im2col	Image Block to Columns

Table 3: Supported operations for encrypted tensors

A.3 DOT PRODUCT

Figure 3 shows how an encrypted vector (in gray) can be multiplied with a plain matrix using the method from Halevi & Shoup (2014).

A.4 2D CONVOLUTION

A 2D convolution can be performed using a single matrix multiplication, instead of repeating multiplication on every window. This method is referred to as image block to column convolution, or image to column convolution. Figure 4 shows how a convolution can be performed using this method. It first reorganizes the input matrix into rows representing convolution windows, then performs a dot product with the flattened kernel.

Applying this technique to an encrypted matrix, which is encrypted into a single ciphertext, is not trivial, as reorganizing slots is not that simple. We will need to reorganize the matrix as a preprocessing step before encryption to be ready for convolution. The encrypted-matrix (input image)

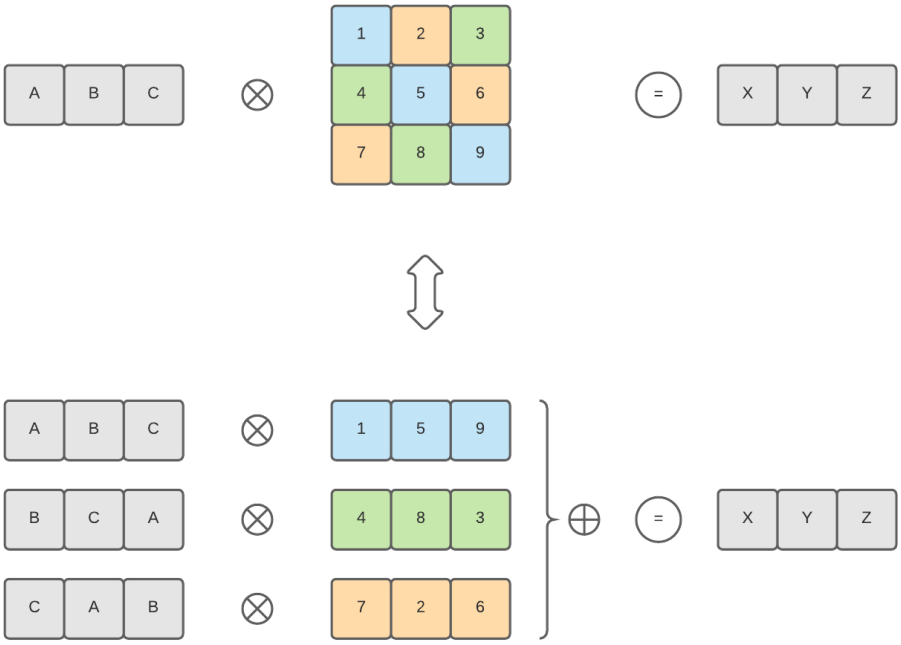

Figure 3: Vector-Matrix Multiplication

with plain-vector (kernel) can be performed with a single element-wise multiplication and a series of rotations and accumulations. Figure 5 and 6 show the steps for doing that. The first shows how the encrypted matrix (colored) is encoded and multiplied with the plain kernel. The second step is to sum different versions of the output that are rotated differently to the left.

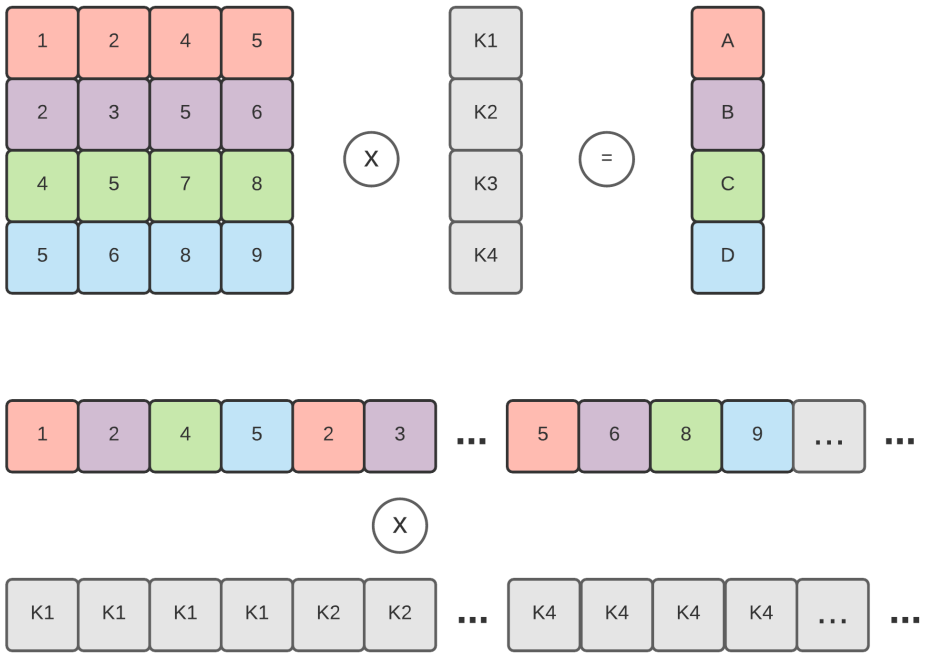

Figure 5: Image to column convolution with CKKS - step 1

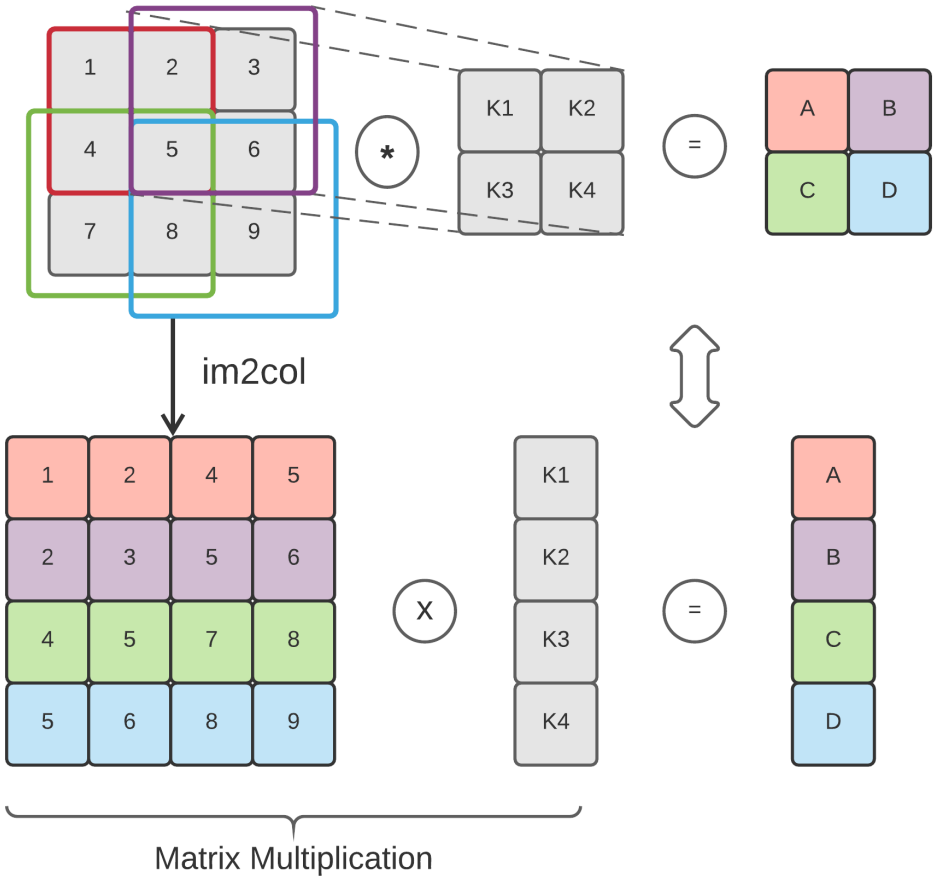

Figure 4: Image to column convolution

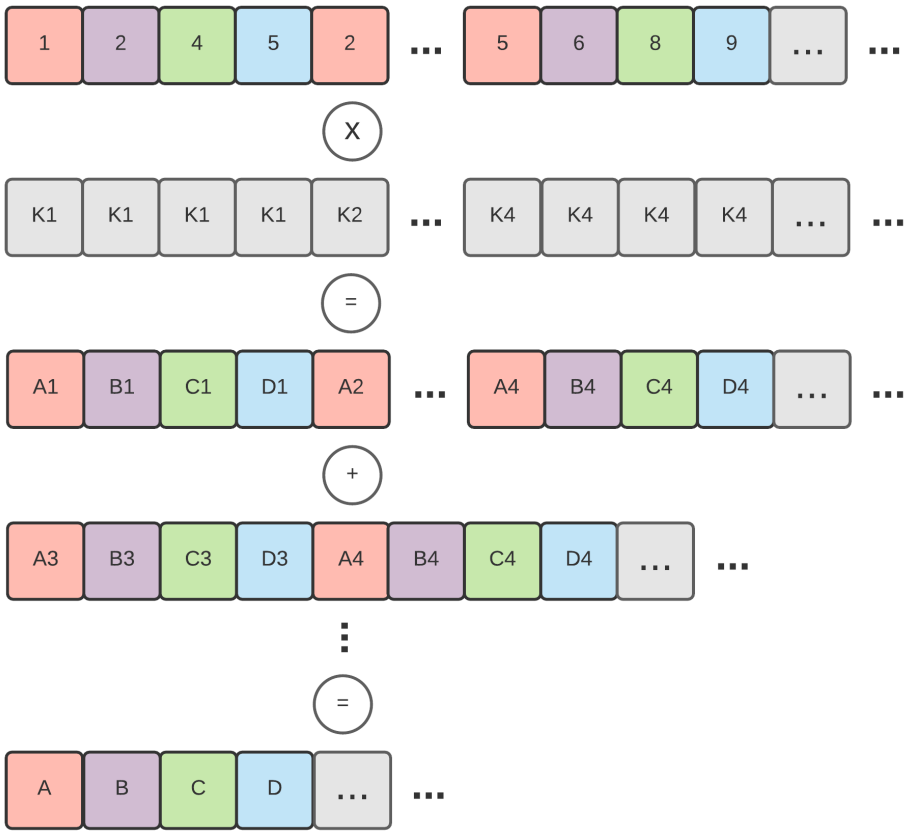

Figure 6: Image to column convolution with CKKS - step 2

A.5 COMPLETE BENCHMARKS

In this section, we present the full evaluation of TenSEAL's operations, ran on the CKKSVector implementation. The benchmarks are measured on an Amazon EC2 c4.2xlarge instance, with 8 vCPUs at 2.9 GHz(Intel Xeon E5-2666 v3 Processor) and 15 GiB memory. They are executed using Ubuntu Server 20.04 and Python 3.8. The measured durations are the average of 5 rounds of testing, with 10 iterations each.

Tables 4 and 5 show the average performance for different arithmetic operations.

Operation	Tensor shape				
	[256]	[1024]	[4096]	[8192]	[16384]
negate	0.07	0.07	0.07	0.13	0.26
square	4.29	4.29	4.29	8.49	17.16
polyval	10.55	10.46	10.51	21.32	42.68

Table 4: Duration in milliseconds for unary operations. The CKKS context is created for polynomial modulus 8192 and coefficient modulus of 200-bits. The polyval benchmark is executed for $2X^2 + X$ polynoms.

Operation	Tensor shape				
	[256]	[1024]	[4096]	[8192]	[16384]
add	0.08	0.08	0.08	0.16	0.31
multiply	4.45	4.34	4.43	8.84	17.75
sub	0.08	0.08	0.08	0.15	0.3
dot	20.15	23.96	28.11	55.94	112.36
add_plain	0.8	0.86	1.07	2.13	4.19
multiply_plain	1.75	1.81	2.03	4.02	7.97
sub_plain	0.8	0.86	1.08	2.14	4.21
dot_plain	17.37	21.36	25.63	51.14	101.82

Table 5: Duration in milliseconds for binary operations. The CKKS context is created for polynomial modulus 8192 and coefficient modulus of 200-bits. For the "_plain" operations, the operand is a PlainTensor of the same shape. For the rest, the operand is an encrypted tensor of the same shape.