Latency Improvement of Homomorphic Encrypted Deep Neural Network Training

Tiago Monteiro[†] Takuya Suzuki[‡] and Hayato Yamana[§]

† ‡ Graduate School of Fundamental Science and Engineering 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

§ Faculty of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

E-mail: † § {tiago, t-suzuki, yamana} @yama.info.waseda.ac.jp

Abstract Machine learning as a service (MLaaS) has become increasingly popular. However, MLaaS has privacy concerns attached. Privacy-preserving machine learning (PPML) solves privacy concerns through cryptographic techniques such as homomorphic encryption (HE). HE enables computations over ciphertexts without decryption. However, HE requires high computational costs and has limited operations. Consequently, many optimization algorithms, key components in PPML training over HE, are hard to be adopted. Therefore, adopting a suitable optimization algorithm for HE is important. This paper aims to decrease the training latency of a homomorphic encrypted deep neural network. We take advantage of Nesterov's accelerated gradient descent and the SIMD computation capabilities of HE.

Keywords: Privacy-preserving machine learning, Homomorphic encryption, Deep neural network training, Nesterov's accelerated gradient, SIMD computation

1. Introduction

Deep learning (DL) has become a valuable method for solving multifaceted problems. Finance, speech/visual recognition, and health sciences are examples of fields influenced by this technology. However, to fully take advantage of DL, two prerequisites exist: 1) users must be capable of selecting adequate DL model and hyperparameters for a specific task; and 2) users must have sufficient computational resources to execute a selected model. Machine Learning as a Service (MLaaS), a cloud-based service, is a solution that provides a layer of abstraction around the two prerequisites needed to take advantage of DL models successfully.

Despite the advantages provided by MLaaS platforms, users with sensitive information such as medical or financial records are concerned by potential data leakages, which are becoming more common. In response to this concern, privacy-preserving machine learning (PPML) ensures that MLaaS parties' resources, such as the user's data or the service provider's neural network model, are secure. With homomorphic encryption (HE) [1,2], we can assure data privacy and usability in a MLaaS pipeline.

Predictions with PPML over HE have achieved good results [3, 4, 5, 6]. Conversely, PPML over HE still struggles to train a DL model with encrypted data, having not yet achieved significant breakthroughs. The computational complexity of the operations involved in the DL's tasks and the high computational cost of HE schemes are the main reasons behind the lack of research effort on this front.

In 2019, Nandakumar et al. [7] was the first to achieve private training of neural networks non-interactively. They demonstrated a neural network that can train a mini-batch of 60 samples in 40 minutes meanwhile achieving an accuracy of 97.8% on the MNIST dataset. However, they used table-lookups to calculate the non-linear functions, severely hindering their work's latency.

In 2020, Lou et al. [8] improved the solutions proposed by Boemer et al. [5] and achieved 98.8% accuracy on the MNIST dataset in 5 epochs, which took 8 days to execute over the prior state-of-the-art system [7]. They utilized CHIMERA [9], a key-switching scheme to solve the issues with BGV-based lookup tables. Although their results improved, they did not present the latency cost of switching between schemes in their results.

The above works provided breakthroughs in non-interactive DL training with HE, but there is still space for latency improvements.

In this work, we further reduce the training latency of a homomorphically encrypted neural network model. Our main contributions are the following:

- We introduce the usage of Nesterov's accelerated descent (NAG) instead of gradient descent (GD). NAG is a promising optimization method because it accelerates training convergence, i.e., decreases the number of training iterations to achieve maximum accuracy compared to GD.

- We take advantage of the SIMD capabilities presented in HE schemes to shorten the homomorphically encrypted DL training latency.

The remainder of this paper is organized as follows: Section 2 presents the required background on neural networks and HE to build our solution. Section 3 shows a brief survey on PPML with HE solutions, highlighting the key achievements by works similar to ours. Section 4 describes our proposed solution; namely, we show how we successfully apply NAG. Section 5 provides experimental results and discussion about the obtained results. Finally, a conclusion is provided in Section 6.

2. Preliminaries

2.1 Fully homomorphic encryption (FHE)

Homomorphic encryption (HE) is a cryptographic technique that allows computations over ciphertexts without leaking its contents, where the results from the computations over ciphertexts are equivalent to the results from the computations over plaintexts. HE allows two operations to be performed on ciphertexts: addition and multiplication.

Presently, fully homomorphic encryption (FHE) is a variant of HE that allows an arbitrary number of additions and multiplications. In an FHE scheme, when a plaintext is encrypted to a ciphertext, a random element, called noise, is added to the ciphertext so that it is harder to reveal the original plaintext. Moreover, the noise in a ciphertext grows with every homomorphic operation. After a number of operations, the noise growth surpasses the ciphertext threshold, which results in rendering the ciphertext unusable.

To overcome the noise accumulation problem, a special operation, called bootstrapping, is used. Bootstrapping comprises a ciphertext re-encryption using a different key; therefore, a new ciphertext with smaller noise than the original ciphertext is obtained. Bootstrapping allows any ciphertext to be evaluated correctly; however, bootstrapping is a time and resource consuming operation.

In this study, we adopt the CKKS scheme [2] , which can perform approximate computations on fixed-point real numbers, bootstrapping, and single-instruction multiple-data (SIMD) computations on the ciphertext.

2.2 Deep learning model

Neural networks are multi-layered, weighted, and directed graphs where input nodes are connected to output nodes through several hidden layers. At the core of neural networks' operations, matrix multiplications and dot products are performed. In deep neural network training, forward and backpropagation stages exist. The training phase aims to learn parameters that minimize the values from a pre-defined loss function.

$$H^l = \phi(W^l \cdot Z^{l-1}) \qquad (2.1)$$

A network with L layers is traversed sequentially from the input to the output layer. Equation (2.1) describes the forward propagation at the l-th layer. The output of the l-th layer Z^l is obtained by applying a non-linear function ϕ to the matrix product of the weights from the current layer, W^l, with the output from the $(l-1)$-th layer, Z^{l-1}.

$$W_{t+1} \coloneqq W_{t+1} - \alpha \frac{\partial \mathcal{L}}{\partial W_{t+1}} \qquad (2.2)$$

Equation (2.2) describes the gradient descent algorithm at t-ath iteration/step with learning rate α and gradient $\frac{\partial \mathcal{L}}{\partial W_{t+1}}$, of the loss function $\mathcal{L}$. GD is an optimization algorithm aiming to find the local minimum of $\mathcal{L}$. During the backpropagation stage, a network is traversed sequentially, in reverse order, from the output to the input layer. The loss function result, calculated at the end of the forward propagation, is passed backward so that the weights from each layer are updated according to their respective influence in the error result.

2.3 Nesterov's Accelerated Gradient (NAG)

The training phase is utilized to find the weights of the hidden and output layers that minimize the values of a loss function. GD follows the negative gradient of an objective function to locate the local minimum of a loss function. However, a GD's limitation arises when the negative gradient direction rapidly oscillates: the zig-zagging problem. Zig-zagging slows down the loss function minimization, i.e., unnecessary iterations are made to minimize a loss function adequately.

A solution to the zig-zagging problem is the momentum method [10]. Yurii Nesterov proposed one of the first GD schemes using momentum in 1983 [11]. In 2013, Sutskever et al. [12] popularized its application in neural network training. The update rules take the following form shown in Equation (2.3) and Equation (2.4). Equation (2.3) describes how the next step in the algorithm is taken, and Equation (2.4) describes the update of the weight matrix

W.

In Equation (2.3), starting with $V_0 = 0$, the algorithm iterates for $t = 1,2,\cdots$:

$$V_{t+1} = \mu V_t - \alpha \nabla \mathcal{L}(W_t + \mu V_t) \tag{2.3}$$

$$W_{t+1} = W_t + V_{t+1} \tag{2.4}$$

, where α is the learning rate; $\nabla \mathcal{L}$ is the direction of the descent; μ is the momentum coefficient; and V is the velocity matrix, accumulating previous gradient values.

The momentum term μV_t scales the value of previous gradients, determining how much the previous gradients influence the current value. μV_t makes the algorithm move faster, i.e., take bigger steps, when updates are consistently small and in the same direction, and move slow, i.e., take smaller steps, when the gradient direction is significantly oscillating. The gradient term $\nabla \mathcal{L}(W_t + \mu V_t)$ determines the descent direction. Compared with Equation (2.2) the gradient term from Equation (2.3) includes μV_t, which corrects the step taken at μV_t if the update is poor, thus giving more stability to NAG.

2.4 Least squares approximation of the sigmoid function

Although NAG performs better than GD, other technical problems remain in implementing our solution. In Equation (2.1), the non-linear function ϕ is the biggest evaluation obstacle, since HE schemes only allow the evaluation of additions and multiplications.

To make use of the sigmoid function as our non-linear function, we can adopt one of two options: lookup tables or function approximation. Lookup tables have been used to train DNNs in [7, 8]. One of the disadvantages of lookup tables is the low resolution of its values. The low value resolution is caused by the limited number of entries in the lookup table. Moreover, the time preparing the table lookup is long. Another solution is approximating the non-linear function [14]. Taylor polynomials have high accuracy values over small ranges, however, the error grows rapidly outside the specified range.

Therefore, we use the least squares fitting polynomial to approximate the sigmoid function. Least squares provides a sufficient approximation with a given interval. Figure 1 plots the original sigmoid function, its least squares fitting polynomial (with degree 3), and its Taylor approximation (of degree 3) within the interval $[-8, 8]$. It can be observed that the Taylor approximation provides an accurate approximation only around a small range, while the least squares fitting polynomial provides a more accurate approximation across a wider range.

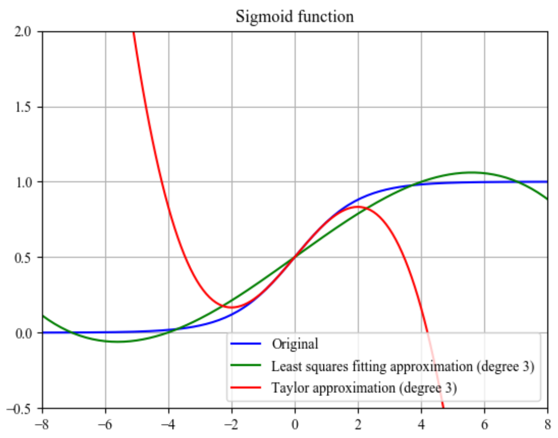

Figure 1 – Original sigmoid (blue), least squares fitting approximation (green), and Taylor approximation (red)

3. Related Work

This section describes recent works that used non-interactive HE neural networks. We selected the non-interactive HE approach for our neural network because it resembles the best MLaaS platforms. MLaaS platforms do not need the user to be online meanwhile the platform performs computations. Figure 2 outlines the common non-interactive training process. An advantage of non-interactive training is that a user can go offline after the user sends the data to the service provider. We explain the flow of the non-interactive training as follows:

1. A user encrypts data with the user's public key.
2. The user shares the encrypted data and the public key to the service provider.
3. The service provider initiates training with the encrypted data.
4. After the training, the service provider obtains the encrypted DL model parameters that are ideal for performing predictions. Note that the service provider can train the model without learning anything about the user's data or the resulting model parameters that have been learned.
5. The service provider sends the encrypted DL model parameters to the user.
6. The user decrypts the encrypted DL model parameters and obtains the unencrypted DL model parameters. Note that only the user can decrypt the encrypted data since only the user has access to the private key.

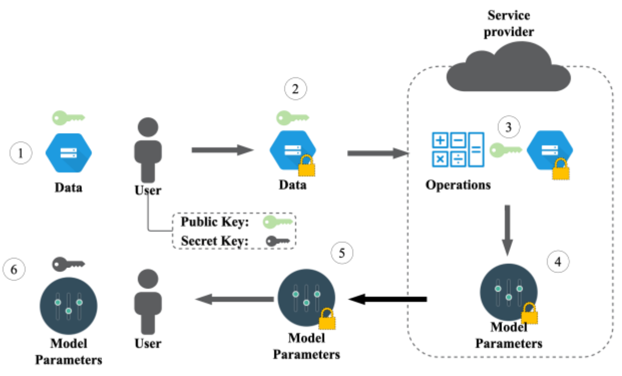

Figure 2 – Neural network training system model

Table 1 – Recent works on training DNNs over HE

Paper	Scheme	Dataset	Acc (%)	#Epoch	Time
Nandakumar et al. [7]	BGV [13]	MNIST	97.8	50	13.4 years
Low et al. [8]	CHIMERA [9]	MNIST	98.8	5	8 days

Table 1 summarizes recent research on DL training with FHE. The problem with existing studies is that they do not achieve optimal latency results for training. Low et al. [8] improve previous state-of-the-art results [7] by achieving higher accuracy within fewer training iterations, but (1) they use the same optimization algorithm as [7] to perform training, and (2) they do not focus on packing the input data effectively to decrease latency. Thus, a new method is needed to obtain enough accuracy in a shorter time.

4. Proposed method

In this work, we propose to reduce the latency of a homomorphically encrypted deep neural network with HE, using the non-interactive HE approach. Our methodology combines the optimization algorithm NAG to decrease the number of training iterations needed to achieve maximum accuracy of a model and an efficient ciphertext packing method that takes advantage of CKKS's SIMD capabilities.

This section is organized as follows: we explain how we adapt NAG to our deep neural network in Section 4.1, and in Section 4.2 we describe the ciphertext packing method utilized in our experiment.

4.1 Evaluation of NAG

Algorithm 1 shows how the encrypted training with NAG is executed. First, the service provider receives two sets of ciphertexts, $|X|$ and $|Y|$, corresponding to the encryption of the training set and its labels, respectively. The service provider initializes the encrypted weights $|W|$ randomly, and sets the encrypted momentum values $|V|$ to zero. For each layer l, the output of the layer, $|H^l|$, is calculated by the polynomial approximation of the sigmoid function ϕ'. The input of ϕ' is the result from the matrix multiplication between the encrypted weights at layer l, $|W^l|$, and the output from the previous layer $|H^{l-1}|$. Note that the values of $|H^0|$ are assgined to $|X|$. After completing the traversal of the neural network from the input to the output layer, the error result $\mathcal{J}$ is calculated by the mean squared error between $|H^L|$ and $|Y|$.

With the obtained error values, the algorithm traverses each layer sequentially in reverse order, updating each layer's weights with regard to the error value. During the updates at each layer, the encrypted momentum values are calculated using Equation (2.3) before updating the weights using Equation (2.4).

It is expected for NAG to achieve maximum accuracy in fewer steps compared to GD. Concretely, after t iterations, NAG will have a rate of convergence of $O(1/t^2)$ compared to $O(1/t)$ in GD. However, NAG has higher computational complexity in each hidden layer compared to GD. The increased complexity from NAG is caused by additional calculations. Specifically the scaled momentum values, and the gradients of the weights values with the respective scaled momentum values.

Algorithm 1: Encrypted NAG training

```
 1: Input: encrypted training samples $|X|$, encrypted
    training labels $|Y|$, learning rate $\alpha$, momentum
    parameter $\mu$, the number of iterations $T$, number
    of layers $L$, and a polynomial approximation of
    sigmoid function $\phi'$
 2: Output: encrypted weights $|W^1|, |W^2|, \cdots, |W^l|$
 3: $|W^1|, |W^2|, \cdots, |W^l| \leftarrow$ Random initialization
 4: $|V^1|, |V^2|, \cdots, |V^l| \leftarrow$ Initialize to zero
 5: $|H^0| \coloneqq |X|$
 6: for $t \leftarrow 1$ to $T$ do
 7:     for $l \leftarrow 1$ to $L$ do
 8:         $|H^l| = \phi'\left(|H^{l-1} \cdot W^l|\right)$
 9:     end for
10:     $\mathcal{J} \leftarrow Loss$
11:     for $l \leftarrow L$ to $1$ do
12:         $|V_{t+1}^l| \leftarrow |\mu V_t^l| - |\alpha \nabla \mathcal{J}(|W_t^l + \mu V_t^l|)|$
13:         $|W_{t+1}^l| \coloneqq |W_t^l + V_{t+1}^l|$
14:     end for
15: end for
16: return $|W^1|, |W^2|, \cdots, |W^l|$
```

4.2 Ciphertext packing

As mentioned in Section 2.1, we utilize the SIMD capabilities of HE. Specifically, we take advantage of HE's packing capabilities. Ciphertext packing consists of encrypting a vector of n plaintexts into a single ciphertext. Each plaintext element is stored separately in a ciphertext slot, and ciphertext operations (additions and multiplications) are performed slot-wise, i.e., element-wise.

In this work, we adapt the packing algorithm ideas of Han et al. [15] and Aharoni et al. [16]. Han et al.'s algorithm [15] allows us to partition a matrix of data into smaller sub-matrixes and then encrypt the sub-matrices with multiple ciphertexts. Aharoni et al.'s algorithms [16] extend the matrix vector multiplication algorithm from [15] to include matrix multiplications.

Given that the input data is split into batches, a batch can be considered to have n samples, and each sample to have m features. Thus, a batch X can be represented as a $n \times m$ matrix. Then, X is divided into multiple $f \times g$ sub-matrices $X_{i,j}$ for $0 \leq i < \lceil n/f \rceil$ and $0 \leq j < \lceil m/g \rceil$. The sub-matrices are supposed to be packed into a single ciphertext; therefore, f and g are set to utilize the maximum number of $slots$ in a ciphertext.

Figure 3 shows an example of a training dataset containing 3 samples with 6 features. The dataset is partitioned into 4 sub-matrices and then encrypted by ciphertexts containing 8 slots each. Note that the slots that are not utilized (light blue squares) are set to zero.

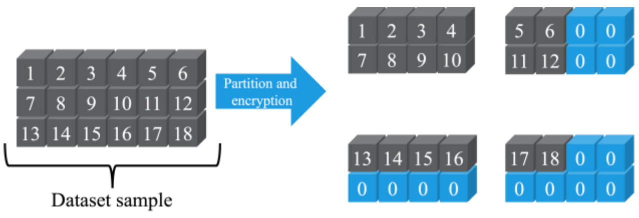

Figure 3 – Partition of a matrix (left) followed by encryption through multiple ciphertexts

4.3 Ciphertext multiplication

Matrix multiplications are some of the most time-consuming operations in the HE. To efficiently perform matrix multiplications, we utilized the packing scheme mentioned in the previous section.

To exemplify the underlying steps in matrix multiplication, we utilize an example of how matrix-vector multiplication is performed on Figure 4. The following steps can be generalized to matrix multiplications [16]. We pack and encrypt a matrix $M \in \mathbb{R}^{3\times6}$ and a vector $v \in \mathbb{R}^{1\times6}$. Note that the number of features in M must match the number of columns in v. The values in v are replicated (dark blue squares) to match the dimensions from M.

Once the data are packed, we compute element-wise multiplication over the corresponding ciphertexts. The results (yellow squares) are obtained by repeating the rotation and addition operations $\log_2(\#\,slots)$. Note that the values marked with "*" (light grey squares) are unknown/garbage values. The unknown values can be cleaned by multiplication with zero vectors.

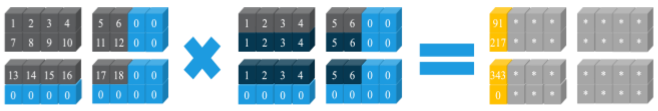

Figure 4 – Matrix vector multiplication

5. Experimental evaluation

We conducted experiments to evaluate NAG against GD utilizing the MNIST [17] dataset. We compared NAG and GD using two packing algorithms: the packing algorithm from Nandakumar et al. and the packing algorithm from Han et al. and Aharoni et al.

5.1 Dataset

The MNIST dataset for handwritten digit recognition consists of 60,000 training samples and 10,000 testing samples. Each image contains a handwritten number between 0 and 9. The images of the MNIST dataset have a gray-scale 28x28 pixel resolution, with one handwritten digit located at the center of the image. Each image is labeled with a class ranging from 0 to 9.

5.2 Network architecture

The neural network in our experiment followed the same architecture used by Nandakumar et al.'s NN2 [7], which consists of a 3 layer fully-connected network with 64 neurons on the input layer, 32 neurons on the first hidden layer, 16 layers on the second hidden layer, and 10 neurons on the output layer.

5.3 Evaluation method

We implemented the homomorphically encrypted deep neural network over HE using HEAAN's [18] CKKS scheme.

When performing the experiments, two configurations were prepared for the target neural network: baseline and proposed. The baseline configuration utilizes the packing and matrix multiplication algorithms used by Nandakumar et al., and the proposed configuration uses the partitioned packing and matrix multiplications algorithms described in Section 4.2. The configurations' CKKS parameters are shown in Table 2. For both configurations, we set the rescaling factor to 23, and the modulus size to 600. For the baseline configuration, we set the number of ciphertext slots to 64, and for the proposed configuration, we set the number of slots to 2,048.

Similar to Nandakumar et al., each image from the MNIST dataset was compressed by cropping the central 24x24 pixels and rescaling by a factor of 1/3 using bicubic interpolation, thus obtaining an 8x8 pixel image representation.

We evaluated the execution time of the training algorithm, the generation of weights, and the matrix multiplication operation. Note that the execution time was averaged by three continuous executions after the first execution.

We used a server which had a Xeon Platinum 8280 (2.7 GHz) with 56 cores, and 1.5TB of main memory for training over HE.

Table 2 – CKKS parameters for the experiment

Packing algorithm	Rescaling factor	Level	Modulus size	# slots
Baseline	23	34	600	64
Proposed	23	34	600	2,048

5.4 Evaluation results

Table 3 shows the training execution time results over the MNIST dataset. The measurements were performed using 56 threads with NTL for one training iteration. One training iteration is enough to estimate the overall execution time of the training algorithm.

When using NAG, 1 training iteration took 8.35 minutes, 5.52 times faster than the baseline method using NAG. When using GD, 1 training iteration took 7.50 minutes, 4.54 times faster than the baseline method.

Table 4 shows the execution time encrypting the generated weights. The proposed algorithm is has lower execution time than the baseline algorithm because the proposed method's weight encryption algorithm is the same as the encryption algorithm of the inputs. In contrast, the baseline method had each element on a weight matrix encrypted individually, and the input data is encrypted by row. Moreover, the proposed packing algorithm has smaller latency when performing matrix multiplications in different layers.

Table 5 shows the latency of matrix multiplications with one mini-batch. The proposed packing algorithm performed 9.8 times faster matrix multiplications on hidden layer 1 compared to the baseline. This is because the proposed packing algorithm requires 1 multiplication and at most $\log_2(\# slots)$ rotations during matrix multiplications. In contrast, the baseline packing algorithm requires at most n^2 multiplications when performiong matrix multiplications. In comparison to ciphertext rotations, homomorphic multiplications are computationally more expensive rotations, thus, the lesser multiplications, the lower the latency.

Table 3 – Execution time of 1 training iteration with 1 mini-batch (containing 64 samples)

Optimization algorithm	Baseline	Proposed
	Training latency (min)	
GD	34.07	7.50
NAG	46.15	8.35

Table 4 – Execution time of weight encryption

Packing algorithm	Hidden layer 1 (sec)	Hidden layer 2 (sec)	Output layer (sec)
Baseline	96.34	22.02	10.86
Proposed	23.30	4.72	4.68

Table 5 – Execution time of matrix multiplications

Packing algorithm	Hidden layer 1 (sec)	Hidden layer 2 (sec)	Output layer (sec)
Baseline	392.85	68.69	38.57
Proposed	40.05	26.72	20.29

6. Conclusion

This paper proposed a set of methods to decrease the training latency of a homomorphically encrypted deep neural network. Nesterov's accelerated gradient (NAG) descent was used instead of gradient descent (GD) to decrease the number of iterations and to achieve maximum accuracy in the training phase; and the packing algorithms from Han et al. and Aharoni et al. were adapted into our solution.

The experimental evaluation was conducted on a compressed MNIST dataset, confirming that the training latency was lower than a neural network packing the data by rows. Compared to the baseline packing algorithm, the proposed packing algorithm can decrease the neural network's latency in weight generation and matrix multiplication.

Future work includes 1) conducting the accuracy comparison between GD and NAG after the homomorphic training, and 2) evaluating the training latency of NAG with the proposed packing solution on the MNIST dataset without compression

References

[1] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. Proceedings of the 41st annual ACM symposium on Theory of computing, pp. 169–178.

[2] Cheon, J. H., Han, K., Kim, A., Kim, M., and Song, Y. (2018). Bootstrapping for approximate homomorphic encryption. Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques , pp. 360–384. Springer, Cham.

[3] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. (2016). Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. International Conference on Machine Learning, pp. 201–210.

[4] Al Badawi, A., Jin, C., Lin, J., Mun, C. F., Jie, S. J., Tan, B. H. M., Nan, X., Khin A. M. M.,and Chandrasekhar, V. R. (2020). Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. IEEE Transactions on Emerging Topics in Computing, 9(3), pp. 1330–1343.

[5] Boemer, F., Lao, Y., Cammarota, R., and Wierzynski, C. (2019). nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. Proceedings of the 16th ACM International Conference on Computing Frontiers, pp. 3–13.

[6] Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., and Mytkowicz, T. (2019). CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 142–156.

[7] Nandakumar, K., Ratha, N., Pankanti, S., and Halevi, S. (2019), Towards Deep Neural Network Training on Encrypted Data. IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2019, pp. 40–48

[8] Lou, Q., Feng, B., Charles Fox, G., and Jiang, L. (2020). Glyph: Fast and accurately training deep neural networks on encrypted data. Advances in Neural Information Processing Systems, 33, pp. 9193–9202.

[9] Boura, C., Gama, N., Georgieva, M., and Jetchev, D. (2020). Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. Journal of Mathematical Cryptology, vol. 14, no. 1, pp.316-338.

[10] Kim, A., Song, Y., Kim, M., Lee, K., and Cheon, J. H. (2018). Logistic regression model training based on the approximate homomorphic encryption. BMC medical genomics, 11(4), pp. 23–31.

[11] Nesterov, Y. E. (1983). A method for solving the convex programming problem with convergence rate O(1/k^2). Doklady Akademii Nauk SSSR , 269(3), pp. 543–547.

[12] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. International conference on machine learning, pp. 1139–1147. PMLR.

[13] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2014). (Leveled) fully homomorphic encryption without bootstrapping. ACM TOCT, vol. 6, no. 3, pp. 1–36.

[14] Baruch, M., Drucker, N., Greenberg, L., and Moshkowich, G. (2022). A Methodology for Training Homomorphic Encryption Friendly Neural Networks. Applied Cryptography and Network Security Workshops. Lecture Notes in Computer Science, vol 13285. Springer, Cham.

[15] Han, K., Hong, S., Cheon, J. H., and Park, D. (2018). Efficient logistic regression on large encrypted data. Cryptology ePrint Archive.

[16] Aharoni, E., Adir, A., Baruch, M., Drucker, N., Ezov, G., Farkash, A., Greenbder, L., Masalha, R., Moshkowich, G., Murik, D., Shaul, H., Soceanu, and Soceanu, O. (2020). HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. arXiv e-prints, arXiv-2011.

[17] LeCun, Y., Burges, C. J., and Cortes, C. (1998). The mnist database. MNIST handwritten digit database. Retrieved February 14, 2023, from http://yann.lecun.com/exdb/mnist/

[18] Snucrypto. (n.d.). *SNUCRYPTO/Heaan*. GitHub. Retrieved February 15, 2023, from https://github.com/snucrypto/HEAAN