A Low-Depth Homomorphic Circuit for Logistic Regression Model Training

Eric Crockett
ericcro@amazon.com
Amazon Web Services

ABSTRACT

Machine learning is an important tool for analyzing large data sets, but its use on sensitive data may be limited by regulation. One solution to this problem is to perform machine learning tasks on encrypted data using homomorphic encryption, which enables arbitrary computation on encrypted data. We take a fresh look at one specific task: training a logistic regression model on encrypted data. The most important factor in the efficiency of a solution is the multiplicative depth of the homomorphic circuit. Two prior works have given circuits with multiplicative depth of five per training iteration. We optimize one of these solutions, by Han et al. [13], and give a circuit with *half* the multiplicative depth per iteration on average, which allows us to perform twice as many training iterations in the same amount of time.

In the process of improving the state-of-the-art circuit for this task, we identify general techniques to improve homomorphic circuit design for two broad classes of algorithms: iterative algorithms, and algorithms based on linear algebra over real numbers. First, we formalize the encoding scheme from [13] for encoding linear algebra objects as plaintexts in the CKKS homomorphic encryption scheme. We also show how to use this encoding to homomorphically compute many basic linear algebra operations, including novel operations not discussed in prior work. This "toolkit" is generic, and can be used in any application based on linear algebra. Second, we demonstrate how generic compiler techniques for loop optimization can be used to reduce the multiplicative depth of iterative algorithms.

KEYWORDS

homomorphic encryption, ckks, logistic regression, model training, linear algebra

ACM Reference Format:
Eric Crockett. 2022. A Low-Depth Homomorphic Circuit for Logistic Regression Model Training. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cloud computing allows anyone–from individuals to enterprises–to migrate on-premise workloads to a third-party. Although it can greatly increase the computing power available to users with limited budgets and computational capability, cloud computing may not be possible in all settings. For example, the European Union's (EU) General Data Protection Regulation (GDPR) regulates data protection and privacy in the EU, and also restricts what information can be transferred outside the EU. Outsourcing computation to the cloud could violate the GDPR if the data to be processed is sensitive and the cloud provider is based outside the EU. One way to maintain data privacy and GDPR compliance is to encrypt the data before sending it to the cloud. By utilizing *homomorphic encryption* (HE), which enables arbitrary computation on encrypted data, the data owner can outsource the computation while ensuring that the cloud provider learns nothing about the sensitive data.

While the use case is simple, actually deploying such a solution is a complex task. The main challenge is designing the homomorphic version of the function to be evaluated. First, the input data must be encoded as homomorphic plaintexts. There are many possible ways to encode the same data, but the encoding will impact the efficiency of the resulting circuit. Next, the function must be implemented using only the native operations (or "gates") of the HE scheme. Since HE schemes only define basic functionality like adding or multiplying plaintexts, it can be challenging to implement high-level operations like rounding, evaluation of non-polynomial functions, and many other operations. Finally, the circuit must be optimized to reduce its *multiplicative depth*, the maximum number of multiplications along any path through the circuit. A circuit's multiplicative depth determines the HE parameters which must be used; lower depth leads to smaller parameters and ciphertexts, which reduces evaluation time and bandwidth.

In this work, we examine these steps for *logistic regression model training*, a basic task in supervised machine learning. The logistic regression (LR) model type is used in myriad applications including genomics [5], medical imaging [19], cancer diagnosis [10], tax compliance [4], credit scores [6], and more. These applications all involve sensitive data, making LR a good target for homomorphic implementation.

Prior work on this problem shows how to encode training data into homomorphic ciphertexts and how to evaluate the training algorithm on encrypted data in a scalable way. Our work improves the model training efficiency, cutting training time in half compared to the best previous result [13]. In the process of improving the state-of-the-art circuit for this task, we identify general techniques to improve homomorphic

circuit design for two broad classes of algorithms: iterative algorithms, and algorithms based on linear algebra over $\mathbb{R}$.

1.1 Contributions

Using the foundation laid out in prior work, we formalize a "toolkit" for linear algebra encodings and operations based on the CKKS HE scheme. The toolkit includes a formal treatment of techniques from [13, 15] as well as novel techniques and observations to increase efficiency and functionality. By formalizing these techniques, we separate them from the specifics of LR training as presented in prior work, which enables these techniques to be used in other applications.[1] This type of formalization is a necessary step along the way to *automating* the creation of homomorphic functions based on linear algebra. For example, our techniques could be included in a framework like RAMPARTS [3] to reduce programmer burden. We demonstrate the benefits of a generalized linear algebra toolkit by using it to reduce communication during LR model training compared to [13].

As a second contribution, we use the novel algorithms and techniques in the toolkit, combined with general techniques from compilers for loop optimization, to give a low-depth circuit for homomorphic LR model training. Designing low-depth circuits for homomorphic evaluation is a crucial part of making HE practical. [13] shows how to evaluate a single iteration of the LR training algorithm using a circuit with multiplicative depth five.[2] We improve this result by giving a circuit for a LR training iteration with an average multiplicative depth of 2.5, effectively doubling the number of training iterations that can be achieved in the same amount of time. This circuit is a drop-in replacement for [13]; in particular it can be combined with the bootstrapping techniques described in that paper to achieve a scalable training algorithm capable of training large models for hundreds of iterations.

We achieve the depth reduction using a combination of *generic* techniques that are applicable to any iterative algorithm. First, we eliminate data dependencies (and therefore reduce depth) by duplicating work. This trades circuit depth for circuit *width*, and results in more overall work. However, this wide circuit turns out to be faster to evaluate in practice because lower circuit depth leads to more efficient operations and because, on highly parallel hardware, wider circuits do not necessarily impact runtime. We also use loop unrolling and pipelining, well-known techniques from compilers, to optimize over multiple iterations of the training algorithm. All of these techniques can be applied to homomorphic circuits for any iterative algorithm.

As with prior works, our circuit is designed for the CKKS approximate HE scheme [9] because of the natural fit between its plaintext space and the inputs to the LR training algorithm. We demonstrate the practical benefits of low-depth, wide circuits using a parallel evaluator based on the SEAL HE library.

[1] An implementation of this toolkit is available as part of the Homomorphic Implementor's Toolkit [14].

[2] Independently, [8] achieved a depth-five circuit for a single iteration of training using a very different approach. To our knowledge, this is the lowest-depth circuit known for this task.

Due to space constraints, this extended abstract omits some details which are present in the full paper [12].

1.2 Related Work

The importance of privacy preserving LR, especially in biomedical applications, has resulted in a significant amount of research on the problem. One of the first attempts to train LR models on encrypted data was [2], which used an additive HE scheme. To compensate for the server's limited computational power, this solution requires significant client precomputation. [11] required the authors to encrypt individual bits of the training data. This solution leaks information about the training data to the model decryptor beyond what can be learned from the model itself. [8] used the w-NIBNAF encoding to encode real numbers as cyclotomic ring elements, and gives a depth-five circuit for computing a single iteration. These solutions all use HE schemes which are exactly homomorphic: homomorphic operations do not incur any plaintext error. However, due to the complexity of the LR training function and the limited functionality provided by native HE operations, they all use various approximations in the computation, meaning the resulting model is different than if the model were trained on plaintexts.

From a performance perspective, the most promising results for LR model training on encrypted data are from a series of papers which utilize the CKKS approximate HE scheme, in which homomorphic operations add a small amount of noise to the plaintext computation. Here, even an exact computation would produce a noisy model; fortunately [2, 7, 11] demonstrate that approximate models can be used effectively for machine learning tasks. In [16] Kim et al. give a circuit for binary classification LR model training using an inefficient linear algebra encoding. This encoding was improved in [15], which also replaces the basic gradient descent step of the training algorithm with a more advanced method call *Nesterov's Accelerated Gradient* (NAG). NAG guarantees faster model convergence, hence fewer training iterations. This solution was the winner of the 2017 Genomic data privacy and security protection competition at the iDash Workshop on Privacy and Security. This work was further improved by Han et al. in [13], who showed how to scale homomorphic LR training for large data sets by utilizing a fundamental HE technique called bootstrapping. They also used *local* circuit optimization techniques (e.g., re-arranging gates within a training iteration) to improve the multiplicative depth of the circuit. All of these works use the CKKS HE scheme [9], which yields efficient operations on floating point data at the cost of producing an approximate result.

2 CKKS HOMOMORPHIC ENCRYPTION SCHEME

This section includes relevant background for the CKKS approximate HE scheme [9]. CKKS is based on the *ring learning with errors* (RLWE) problem, which uses structured polynomial rings called *cyclotomic rings* for efficiency. The plaintext space is $\mathbb{R}^t$, where t is related to the primary CKKS parameter;

see [9] for details. The value t denotes the number of "plaintext slots", on which homomorphic addition and multiplication work component-wise. Throughout this work, we assume that t is a power of two.

Encryption API The CKKS HE scheme has a symmetric-key variant and a public-key variant. Both of these schemes can be used with our application, but we focus on the symmetric-key version for simplicity. We give a simplified encryption API for a security parameter n, and denote the encryption of a plaintext $m \in \mathbb{R}^t$ by $\boxed{m}$.

$$\begin{aligned} k_{\text{eval}}, k_{\text{sk}} &\leftarrow \mathsf{HE.Keygen}(n) \\ \boxed{m} &\leftarrow \mathsf{HE.Encrypt}(k_{\text{sk}}; m) \\ m &\leftarrow \mathsf{HE.Decrypt}(k_{\text{sk}}; \boxed{m}) \end{aligned}$$

Here, k_{sk} is the usual symmetric key, while k_{eval} is called the *evaluation key*. The evaluation key is an abstraction of several different public keys required to perform homomorphic computation, including *relinearization keys* and *Galois keys*. Note that the encoding process from $\mathbb{R}^t$ to a cyclotomic ring element is itself approximate, so unlike most encryption schemes, $\mathsf{HE.Decrypt}(k_{\text{sk}}; \mathsf{HE.Encrypt}(k_{\text{sk}}; m)) \approx m$ rather than exact equality.

Evaluation API In addition to the standard encryption API, HE schemes include additional functions to enable computation on encrypted data. Let x, y, and z be plaintexts in $\mathbb{R}^t$, and $c \in \mathbb{R}$. Let x_i denote the i^{th} component of the plaintext x. The native CKKS operations include:

- $\boxed{x} \oplus \boxed{y} = \boxed{z}$, where $z_i = x_i + y_i$
- $\boxed{x} \oplus y = \boxed{z}$, where $z_i = x_i + y_i$
- $\boxed{x} \oplus c = \boxed{z}$, where $z_i = x_i + c$
- $\boxed{x} \odot \boxed{y} = \boxed{z}$, where $z_i = x_i \cdot y_i$
- $\boxed{x} \odot y = \boxed{z}$, where $z_i = x_i \cdot y_i$
- $\boxed{x} \odot c = \boxed{z}$, where $z_i = c \cdot x_i$
- $\mathsf{Lrot}(k_{\text{eval}}, \boxed{x}) = \boxed{z}$ where z is the left cyclic rotation of x by k positions, i.e., $z_i = x_{i+k \bmod t}$
- $\mathsf{Rrot}(k_{\text{eval}}, \boxed{x}) = \boxed{z}$ where z is the right cyclic rotation of x by k positions, i.e., $z_i = x_{i-k \bmod t}$

Each of these operations also incurs a small error, beyond the standard loss from floating point arithmetic. To simplify notation, we reuse this notation on plaintext operands as well.

Multiplicative Depth and Bootstrapping Like all other "fully homomorphic" encryption schemes, CKKS is more accurately described as a *leveled* HE scheme. This means that it can be parameterized to allow evaluation of any circuit whose *multiplicative depth* is below a certain bound. The multiplicative depth of a circuit is the maximum number of multiplications along any path through the circuit.

In order to transform a leveled HE scheme into a fully homomorphic scheme (i.e., one which can evaluate functions of a priori unbounded multiplicative depth), we need to use *bootstrapping*, the process of homomorphically evaluating the encryption system's decryption circuit to refresh the ciphertext. This option leads to two evaluation strategies for any particular target function: a direct method that does not use bootstrapping, and an iterative evaluation using bootstrapping. This work does not utilize bootstrapping; it focuses on circuit optimization to reduce multiplicative depth, which is important for evaluation with and without bootstrapping.

Parameterization This work focuses on circuit optimization and plaintext operations rather than implementation details. As such, we do not address topics such as the CKKS plaintext scale parameter (used to control the precision of the homomorphic computation), secure instantiations, or other parameters. See [9] for details.

Circuit Design All circuits take plaintexts as inputs and produce plaintext outputs. We do not include "ciphertext maintenance" operations such as rescaling and relinearization. We reuse the ciphertext operations from Section 2 on plaintext arguments to simplify notation.

3 HOMOMORPHIC LINEAR ALGEBRA

This section provides a formal treatment of the novel linear algebra encoding techniques given in [13, 15]. We also extend these techniques with algorithms for homomorphic matrix-matrix multiplication.[3]

Notation We denote a column vector as $\vec{b}$, and denote row vectors as the transpose of a column vector, i.e., $\vec{b}^\intercal$. We will encode these objects as CKKS plaintexts, which we denote by a tuple, e.g., $(a, b, c, d) \in \mathbb{R}^t$. Throughout this paper, we let $t = 2^k$ denote the number of CKKS plaintext slots.

3.1 Basic Encoding

All vectors and matrices, regardless of their dimension or shape, will be encoded as one or more *encoding units*, where an encoding unit is an $m \times n$ matrix such that $m \cdot n = t$. Since $t = 2^k$, note that the encoding unit will *always* have two-power dimensions. We denote the encoding of an object B into a CKKS plaintext relative to an an $m \times n$ encoding unit by $\langle B \rangle^{[m \times n]}$, where we omit the encoding unit when it is clear from context.

$$\text{Define the matrix } A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}, \text{column vector } \vec{x} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix},$$

and row vector $\vec{y}^\intercal = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

Matrix Encoding Assume that we want to encode a matrix which has exactly as many elements as there are plaintext slots for the chosen CKKS parameters, i.e., we wish to encode a matrix in $\mathbb{R}^{m \times n}$ where $m \cdot n = t$ and m, n are powers of two. In this case, we choose the encoding unit to coincide with the matrix dimension, and encode the matrix in row-major order, so for $t = 8$, $\langle A \rangle^{[2 \times 4]} = (1, 2, 3, 4, 5, 6, 7, 8)$.

[3] Note that the encodings discussed in this section actually constitute a *second* layer of encoding during encryption, since we treat the CKKS encoding from a vector of real numbers to a cyclotomic ring element as implicit. See [9] for details.

Vector Encoding We encode a column vector $\vec{x} \in \mathbb{R}^n$ with respect to a single $m \times n$ encoding unit by first creating a $m \times n$ matrix where each row is $\vec{x}^\intercal$, and then encoding the matrix as above. For example, the encoding of $\vec{x}$ is

$$\left\langle \vec{x} \right\rangle^{[2\times4]} = \left\langle \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right\rangle^{[2\times4]} = (1,-1,1,-1,1,-1,1,-1).$$

Similarly, we encode $\vec{b}^\intercal \in \mathbb{R}^m$ as [the encoding of] an $m \times n$ matrix where each column is $\vec{b}$. For example, the encoding of $\vec{y}^\intercal$ is

$$\left\langle \vec{y}^\intercal \right\rangle^{[2\times4]} = \left\langle \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right\rangle^{[2\times4]} = (1,1,1,1,0,0,0,0).$$

3.2 Linear Algebra Operations

When combined with the encodings given above, the component-wise operations induced by $\oplus$ and $\odot$ lead to a natural homomorphic implementation for many linear algebra operations. Let $A, B \in \mathbb{R}^{m\times n}$, $c \in \mathbb{R}$, $\vec{x}, \vec{y} \in \mathbb{R}^n$, $\vec{w}^\intercal, \vec{z}^\intercal \in \mathbb{R}^m$, and define $\mathbf{1}_\bullet$ as the matrix/vector of all 1s whose shape is the same as $\bullet$. The following equations show how homomorphic operations induce simple linear algebra operations on encoded objects, all encoded with respect to the $m \times n$ encoding unit. Note that we do not encode the constant c because we assume this value is public, and therefore is never represented as an encrypted value. On the other hand, matrices and vectors are always encoded, even if not encrypted.

- $\langle A \rangle \oplus \langle B \rangle = \langle A + B \rangle$
- $\langle A \rangle \oplus c = \langle A + c \cdot \mathbf{1}_A \rangle$
- $\langle A \rangle \odot \langle B \rangle = \langle A \circ B \rangle$
- $c \odot \langle A \rangle = \langle c \cdot A \rangle$
- $\langle \vec{x} \rangle \oplus \langle \vec{y} \rangle = \langle \vec{x} + \vec{y} \rangle$
- $\langle \vec{x} \rangle \oplus c = \langle \vec{x} + c \cdot \mathbf{1}_{\vec{x}} \rangle$
- $\langle \vec{x} \rangle \odot \langle \vec{y} \rangle = \langle \vec{x} \circ \vec{y} \rangle$
- $c \odot \langle \vec{x} \rangle = \langle c \cdot \vec{x} \rangle$
- $\langle \vec{w}^\intercal \rangle \oplus \langle \vec{z}^\intercal \rangle = \langle \vec{w}^\intercal + \vec{z}^\intercal \rangle$
- $\langle \vec{w}^\intercal \rangle \oplus c = \langle \vec{w}^\intercal + c \cdot \mathbf{1}_{\vec{w}^\intercal} \rangle$
- $\langle \vec{w}^\intercal \rangle \odot \langle \vec{z}^\intercal \rangle = \langle \vec{w}^\intercal \circ \vec{z}^\intercal \rangle$
- $c \odot \langle \vec{w}^\intercal \rangle = \langle c \cdot \vec{w}^\intercal \rangle$

Matrix/vector multiplication is more complex, and is covered in the following sections.

3.3 Row Vector/Matrix Products

We now turn to the fundamental linear algebra task of multiplying a row vector $\vec{x}^\intercal \in \mathbb{R}^m$ by a matrix $A \in \mathbb{R}^{m\times n}$. Homomorphically, the inputs are an encoded row vector and an encoded matrix, and the output should be an encoded row vector (all with respect to an $m \times n$ encoding unit). We do not quite achieve this; instead we will compute the *transpose* of the desired result, a column vector encoded with an $m \times n$ unit.

The homomorphic computation is a two-step process. The first step multiplies each row by the appropriate scalar in a single homomorphic operation, while the second step sums the scaled rows *and* ensures that the output is the encoding of (the transpose of) this sum.

When m is a power of two, Algorithm 1 gives an algorithm SumRows (first described in [16]) that outputs the encoding of a column vector which is the (transpose of) the sum of the rows of the input matrix, e.g.,

$$\mathsf{SumRows}\left(\left\langle \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right\rangle\right) = \left\langle \begin{bmatrix} a+c & b+d \\ a+c & b+d \end{bmatrix} \right\rangle = \left\langle \begin{bmatrix} a+c \\ b+d \end{bmatrix} \right\rangle.$$

Since it only involves rotating and adding plaintexts, we can also view Algorithm 1 as a homomorphic circuit with multiplicative depth zero.

Algorithm 1 SumRows: summation of matrix rows

Input: $\langle A \rangle^{[m\times n]}$ for $A \in \mathbb{R}^{m\times n}$ and m, n powers of two
Output: An n-dimensional column vector encoded with an $m \times n$ unit

```
R = ⟨A⟩
for 0 ≤ i < log_2 m do
    R = Lrot_{n·2^i}(R) ⊕ R
end for
return R
```

We formalize this algorithm with the following fact:

Fact 3.1. *Let $\vec{x}^\intercal \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m\times n}$. Then* $\mathsf{SumRows}(\langle \vec{x}^\intercal \rangle \odot \langle A \rangle) = \langle (\vec{x}^\intercal \cdot A)^\intercal \rangle = \langle A^\intercal \cdot \vec{x} \rangle$*, where all encodings are relative to an $m \times n$ encoding unit.*

The SumRows instruction is also a linear map:

Fact 3.2. *Let $A, B \in \mathbb{R}^{m\times n}$. Then*

$$\mathsf{SumRows}(\langle A \rangle) \oplus \mathsf{SumRows}(\langle B \rangle) = \mathsf{SumRows}(\langle A \rangle \oplus \langle B \rangle),$$

where all encodings are relative to an $m \times n$ encoding unit.

3.4 Column Vector/Matrix Products

Multiplying a matrix by a column vector is similar: we multiply the encoded matrix and encoded column vector component-wise, then sum the columns of the product. Algorithm 2 defines the SumCols algorithm (first described in [16]) for summing the columns of a matrix and outputting the result as an (encoded) row vector.

Algorithm 2 SumCols: summation of matrix columns

Input: $\langle A \rangle^{[m\times n]}$ for $A \in \mathbb{R}^{m\times n}$ and m, n powers of two
Output: An m-dimensional vector row encoded with an $m \times n$ unit

```
R = ⟨A⟩
for 0 ≤ i < log_2 n do
    R = Lrot_{2^i}(R) ⊕ R
end for
D ∈ R^{m×n} = {d_{i,j}}, where d_{i,j} = 1 if j = 0 and 0 otherwise.
R = R ⊙ ⟨D⟩^{[m×n]}
for 0 ≤ i < log_2 n do
    R = Rrot_{2^i}(R) ⊕ R
end for
return R
```

We formalize this algorithm with the following fact:

Fact 3.3. *Let $\vec{x} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m\times n}$. Then* $\mathsf{SumCols}(\langle A \rangle \odot \langle \vec{x} \rangle) = \langle (A \cdot \vec{x})^\intercal \rangle = \langle \vec{x}^\intercal \cdot A^\intercal \rangle$*, where all encodings are relative to an $m \times n$ encoding unit.*

The SumCols instruction is also an linear map:

Fact 3.4. *Let $A, B \in \mathbb{R}^{m\times n}$. Then*

$$\mathsf{SumCols}(\langle A \rangle) \oplus \mathsf{SumCols}(\langle B \rangle) = \mathsf{SumCols}(\langle A \rangle \oplus \langle B \rangle),$$

where all encodings are relative to an $m \times n$ encoding unit.

This simple identity has major performance implications: it turns out that while cyclic rotation does not consume any HE levels, it is still the most computationally intense operation. This identity allows us to replace multiple calls to SumCols (or SumRows, using Fact 3.2) with a single call.

3.5 Encoding Objects of Arbitrary Dimension

So far, we have described how to operate on $m \times n$ matrices, m-dimensional row vectors, and n-dimensional column vectors, where m and n match the encoding unit. In this section, we extend these operations to include vectors and matrices of arbitrary dimensions. These techniques were used in prior work [13, 15, 16]; we formalize them here.

3.5.1 *Encoding Small Objects.*

Encoding Small Matrices Given an $f \times g$ matrix A where $f \cdot g <= t$ and f, g are arbitrary (i.e., not necessarily powers of two), m, n such that $f \leq m, g \leq n$, we can embed A with respect to an $m \times n$ encoding unit by extending each row of the matrix with $n - g$ 0s and then add $m - f$ all-0 rows below. To encode the matrix A, we first embed into an $m \times n$ matrix as above, and then encode that matrix using the basic row-major encoding: $\langle A \rangle = \left\langle \begin{bmatrix} A & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \right\rangle$. Note that when $f = m$ and $g = n$, this encoding coincides with the basic matrix encoding described above. It is simple to verify that the matrix operations still hold with this encoding.

Encoding Small Vectors There is a corresponding generalized encoding for row vectors of dimension $f \leq m$ or column vectors of dimension $g \leq n$: first pad the vector with zeros to length m or n, respectively, and then use the normal vector encoding. It is easy to verify that the vector operations still hold with this encoding.

Matrix/Vector Multiplication By pairing these encodings for matrices and vectors, we see that matrix-vector multiplication works as expected: for $A \in \mathbb{R}^{f \times g}$, $\vec{b} \in \mathbb{R}^g$ and $\vec{c}^\intercal \in \mathbb{R}^f$, $\mathsf{SumCols}(\langle A \rangle \odot \left\langle \vec{b} \right\rangle) = \left\langle \vec{b}^\intercal \cdot A^\intercal \right\rangle$, and $\mathsf{SumRows}(\langle \vec{c}^\intercal \rangle \odot \langle A \rangle) = \langle A^\intercal \cdot \vec{c} \rangle$, where all encodings are with respect to an $m \times n$ encoding unit.

3.5.2 *Encoding Large Objects.*

Encoding Large Matrices Given an $f \times g$ matrix A for *arbitrary* f, g (i.e., not necessarily powers of two), choose an arbitrary $m \times n$ encoding unit. We can embed A into (several) plaintexts by dividing it into $m \times n$ submatrices and encoding each piece independently. Thus, $\langle A \rangle = \{\langle A \rangle_{i,j}\}_{\substack{0 \leq i < \lceil f/m \rceil \cdot \\ 0 \leq j < \lceil g/n \rceil}}$.

It is straightforward to extend matrix operations to this encoding by applying them to each encoded submatrix, i.e., for $A, B \in \mathbb{R}^{f \times g}$ and $c \in \mathbb{R}$:

- $\langle A \rangle \oplus \langle B \rangle = \{C_{i,j}\}$, where $C_{i,j} = \langle A \rangle_{i,j} \oplus \langle B \rangle_{i,j}$
- $\langle A \rangle \oplus c = \{C_{i,j}\}$, where $C_{i,j} = \langle A \rangle_{i,j} \oplus c$
- $\langle A \rangle \odot \langle B \rangle = \{C_{i,j}\}$, where $C_{i,j} = \langle A \rangle_{i,j} \odot \langle B \rangle_{i,j}$
- $\langle A \rangle \odot c = \{C_{i,j}\}$, where $C_{i,j} = c \odot \langle A \rangle_{i,j}$

Encoding Large Vectors The same technique can be extended to a row (or column) vector $\vec{b}$ of arbitrary dimension f (or g) by zero-extending the vector to a multiple of m (n) and then dividing the vector into $\lceil f/m \rceil$ ($\lceil g/n \rceil$) chunks $\{\vec{b}_i\}$ of size m (n). Each $\vec{b}_i$ is then encoded with an $m \times n$ encoding unit as usual.

Matrix/Vector Multiplication We can also extend matrix/vector multiplication to work for large objects. For $A \in \mathbb{R}^{f \times g}, \vec{b} \in \mathbb{R}^g$,

$$\left\langle \vec{b}^\intercal \cdot A^\intercal \right\rangle = \{C_i\}, \text{ where } C_i = \mathsf{SumCols}(\sum_j \langle A \rangle_{i,j} \odot \left\langle \vec{b} \right\rangle_j).$$

Similarly, for $A \in \mathbb{R}^{f \times g}, \vec{b} \in \mathbb{R}^f$,

$$\left\langle A^\intercal \cdot \vec{b} \right\rangle = \{C_j\}, \text{ where } C_j = \mathsf{SumRows}(\sum_i \left\langle \vec{b}^\intercal \right\rangle_i \odot \langle A \rangle_{i,j}).$$

Extending the SumRows **and** SumCols **Maps** We can extend the SumRows and SumCols maps to work on matrices which do not have the same dimensions, and which we are therefore not able to add. For example, consider $A \in \mathbb{R}^{f \times g_1}$ and $B \in \mathbb{R}^{f \times g_2}$. $\mathsf{SumCols}(\langle A \rangle^{[m \times n]} \oplus \mathsf{SumCols}(\langle B \rangle^{[m \times n]}$ is well defined as the sum of two f-dimensional vectors, but the sums $A + B$ and $\langle A \rangle \oplus \langle B \rangle$ are not. Nevertheless, we can still define an "linear map" by first summing the horizontal encoding units of $\langle A \rangle$ and $\langle B \rangle$ into a single column of units, then adding them together, and then applying SumCols. The same approach allows us to extend the map onto matrices with the same width and different heights using SumRows.

3.6 Matrix/Matrix Products

Given $A \in \mathbb{R}^{f \times g}$, $B \in \mathbb{R}^{g \times h}$, and $c \in \mathbb{R}$, we can compute the product $c \cdot AB$ in two different ways, either viewing the matrix product as the rows of A times the matrix B, or as the matrix A times the columns B. The mathematical algorithm for matrix/matrix multiplication does not distinguish between these views, but due to the encodings used for homomorphic encryption, these two algorithms accept different inputs.

First we examine Algorithm 3, which takes an encryptions of $A^\intercal$ and B, and multiplies the rows of A by B. Each of the f loop iterations (Algorithm 3) can be performed in parallel. Within each iteration:

- Algorithm 3 involves $\lceil \frac{g}{m} \rceil$ parallel multiplications
- Algorithm 3 involves $\lceil \frac{g}{m} \rceil$ parallel rotations
- Algorithm 3 involves $\log_2(n)$ sequential rotations
- Algorithm 3 involves $\lceil \frac{g}{m} \rceil \cdot \lceil \frac{h}{n} \rceil$ parallel multiplications and $\log_2(m)$ sequential rotations
- Algorithm 3 involves $\lceil \frac{h}{n} \rceil$ parallel multiplications

Overall, given a processor with $\approx \frac{f \cdot g \cdot h}{m \cdot n}$ cores, we can compute MatProd : RowMajor for the cost of three multiplications and $1 + \log_2(m) + \log_2(n)$ rotations.

The full version of the paper also gives an algorithm which views matrix/matrix multiplication as a series of matrix/column-vector products.

Algorithm 3 MatProd : RowMajor: product of matrices viewing first matrix as a set of rows

Input: $\langle A^\intercal \rangle^{[m\times n]}$, $\langle B \rangle^{[m\times n]}$, for $A \in \mathbb{R}^{f\times g}, B \in \mathbb{R}^{g\times h}$, and $c \in \mathbb{R}$
Output: $\langle c \cdot A \cdot B \rangle^{[m\times n]}$

for $0 \le k < f$ **do**
▷ Mask for k^{th} col of $\langle A^\intercal \rangle$
$D \in \mathbb{R}^{m\times n} = \{d_{i,j}\}$, where $d_{i,j} = \begin{cases} 1 & \text{if } j \equiv k \bmod n \\ 0 & \text{otherwise} \end{cases}$
▷ Extract k^{th} col
$A_k = \{A_{k,i}\}_{0\le i<\lceil \frac{g}{m} \rceil}$, where $A_{k,i} = \langle D \rangle^{[m\times n]} \odot \langle A^\intercal \rangle_{i,\lfloor \frac{k}{n} \rfloor}$
▷ Shift to first col
$A_k = \text{Lrot}_{k \bmod n}(A_k)$
▷ Replicate across all cols of the unit
for $0 \le j < \log_2(n)$ **do**
$A_k = \text{Rrot}_{2^j}(A_k) \oplus A_k$
end for
▷ k^{th} row of A times B
$R_k = \text{SumRows}(A_k \odot \langle B \rangle)$
▷ Scaled mask for k^{th} row
$E \in \mathbb{R}^{m\times n} = \{e_{i,j}\}$, where $e_{i,j} = \begin{cases} c & \text{if } i \equiv k \bmod m \\ 0 & \text{otherwise} \end{cases}$
▷ Isolate k^{th} row
$S_k = \{S_{k,j}\}_{0\le j<\lceil \frac{h}{n} \rceil}$, where $S_{k,j} = R_{k,j} \odot \langle E \rangle^{[m\times n]}$
end for
▷ Sum m rows to fill rows of one unit
return $T = \{T_i\}_{0\le i<\lceil \frac{f}{m} \rceil}$, where $T_i = \sum_{k=i\cdot m}^{\min(f-1, i\cdot m+m-1)} S_k$

3.7 Operations with mixed encoding units

All of the linear algebra operations we have seen so far work between objects encoded relative to the same unit. However, Section 3.5 introduces ambiguity into how objects are encoded, which means it is plausible to need to operate on objects with *different* encoding units. The full version of the paper introduces a novel technique for matrix-matrix and matrix-vector multiplication with mixed encoding units, when the operands meet specific dimensional requirements.

4 TRAINING A LOGISTIC REGRESSION MODEL

This section describes the basic binary-classification mini-batch logistic regression training algorithm. In supervised machine learning, pre-classified data is used to train a model, which can then be used to classify samples not used to train the model. We only consider the training phase of this process, in which we use a training algorithm to train a logistic regression model.

The training algorithm is iterative; each iteration integrates more labeled training samples to produce a more accurate model. When training on plaintexts, the training process terminates when the model converges. When training on encrypted data, convergence is hard to detect and too expensive to wait for in most applications. Instead, we fix the number of training iterations in advance to trade off between training time and the resulting model's accuracy. The authors of [13] improve the situation with encrypted training data by modifying the basic training algorithm to use *Nesterov's Accelerated Gradient* (NAG) [18] for the gradient descent step of the optimization process. NAG guarantees faster convergence than basic gradient descent, which means we can train for fewer iterations while still producing a good model. We also adopt NAG for logistic regression model training. The full algorithm is given in Algorithm 4.

Training Data Training data is formatted as *mini-batches*. Each mini-batch is a collection of f data points, each with $g-1$ features, formatted as a matrix $X_i \in \mathbb{R}^{f\times g}$, where the first column of the matrix consists of the labels for each data point (either 0 or 1), and the other columns correspond to a specific feature.[4] Rather than taking the mini-batches X_i as inputs, Algorithm 4 expects tweaked mini-batches Z_i. Let $\vec{x}_{ij}^\intercal = (\ell, \overrightarrow{x'}_{ij}^\intercal \in \mathbb{R}^{f-1})$ denote the j^{th} row of X_i where $\ell \in \{0,1\}$ is the label. Let $\ell' = 2\ell - 1$, and set the j^{th} row of Z_i to $\vec{z}_{ij}^\intercal = (\ell', \ell' \cdot \overrightarrow{x'}_{ij})$.

Training Parameters The training algorithm also takes additional training parameters, such as the number of iterations k and the *learning rate* α, which controls how far to move in the direction of the gradient. These values, as well as the internal constants ϵ_i, are transmitted and used in-the-clear, without being encrypted.

Activation Function The LR training algorithm utilizes the *sigmoid* function $\sigma(x) = \frac{1}{1+\exp(-x)}$. Let $\vec{\sigma}(\bullet)$ denote the application of the sigmoid approximation to each component of the input vector. The sigmoid function is difficult to compute homomorphically, so we instead use a least-squares low-degree polynomial approximation over a target range, following [13, 15, 16]. As in those works, we use a cubic polynomial approximation of the form $c_3 \cdot x^3 + c_1 \cdot x + c_0$.

Algorithm 4 Mini-batch training algorithm

Input: Tweaked mini-batches $Z_i \in \mathbb{R}^{f\times g}$ for $0 \le i < k$, learning rate α
Output: $\vec{v}_k \in \mathbb{R}^g$

$\epsilon_0 = 1$
$\vec{w}_0 = \vec{v}_0 = \vec{0} \in \mathbb{R}^g$
$\gamma = \alpha / f$
for $0 \le i < k$ **do**
$\epsilon_{i+1} = \frac{1+\sqrt{4\epsilon_i^2+1}}{2}$
$\eta_i = \frac{1-\epsilon_i}{\epsilon_{i+1}}$
$\vec{\ell}_i = Z_i \cdot \vec{v}_i$
$\vec{b}_i = \vec{\sigma}(-\vec{\ell}_i)$
$\vec{\Delta}_i = Z_i^\intercal \cdot \vec{b}_i$
$\vec{w}_{i+1} = \vec{v}_i + \gamma \cdot \vec{\Delta}_i$
$\vec{v}_{i+1} = (1-\eta_i)\vec{w}_{i+1} + \eta_i \vec{w}_i$
end for
return $\vec{v}_k$

[4] Note that it is not necessary for each mini-batch to have exactly f data points (e.g., the last mini-batch may have fewer than f data points), so f need not exactly divide the total number of labeled training data points.

5 HOMOMORPHIC LOGISTIC REGRESSION TRAINING: DEPTH 4

The efficiency of logistic regression training on encrypted data is primarily a function of the homomorphic circuit used to implement a single iteration of lines 7—11 of Algorithm 4. This circuit is chained together to achieve multiple iterations of the training loop. In this section, we make a simple observation which allows us to tweak the depth-five circuit given in [13, Figure 3] to make a depth-four circuit. In addition, we use the linear algebra toolkit from Section 3 to decouple the encoding unit from the mini-batch size, providing a more general solution than [13].

In order to focus on the relevant portion of Algorithm 4 and move one step closer to the [13] circuit, we give a "homomorphic-friendly" version of lines 7—11 in Algorithm 5. Note that this version uses the sigmoid approximation σ' rather than the true sigmoid, since the approximation is used for efficient homomorphic evaluation.

Han et al. show how to compute Algorithm 5 on encrypted data with a depth-five circuit[5], shown in Algorithm 6, which is essentially [13, Figure 3].

Algorithm 5 Mini-batch training iteration (Mathematical view)

Input: Tweaked mini-batch $Z_i \in \mathbb{R}^{f \times g}$, $\vec{w}_i$ $\vec{v}_i \in \mathbb{R}^g$, and public parameters γ, η_i
Output: $\vec{w}_{i+1}, \vec{v}_{i+1} \in \mathbb{R}^g$

1: $\vec{\ell}_i = Z_i \cdot \vec{v}_i$
2: $\vec{c}_i = \gamma \cdot Z_i^\intercal \cdot \vec{\sigma'}(-\vec{\ell}_i)$
3: $\vec{w}_{i+1} = \vec{v}_i + \vec{c}_i$
4: $\vec{v}_{i+1} = (1-\eta_i)\vec{w}_{i+1} + \eta_i \vec{w}_i$
5: **return** $\vec{w}_{i+1}, \vec{v}_{i+1}$

Algorithm 6 Mini-batch training iteration (Homomorphic view)

Input: Tweaked mini-batch $\langle Z_i \rangle$ and vectors $\langle \vec{w}_i \rangle$, $\langle \vec{v}_i \rangle$, all encoded with an (arbitrary) $m \times n$ unit
Output: $\langle \vec{w}_{i+1} \rangle$ and $\langle \vec{v}_{i+1} \rangle$, both encoded with an $m \times n$ unit

1: $\langle \vec{\ell}_i^\intercal \rangle = \mathsf{SumCols}(\langle Z_i \rangle \odot \langle \vec{v}_i \rangle)$
2: $M_i = \langle \vec{\ell}_i^\intercal \rangle \odot \langle \vec{\ell}_i^\intercal \rangle \oplus \frac{c_1}{c_3}$
3: $\langle \vec{c}_i \rangle = \mathsf{SumRows}(((-\gamma c_3 \odot \langle Z_i \rangle) \odot \langle \vec{\ell}_i^\intercal \rangle) \odot M_i \oplus (\gamma c_0 \odot \langle Z_i \rangle))$
4: $\langle \vec{w}_{i+1} \rangle = \langle \vec{v}_i \rangle \oplus \langle \vec{c}_i \rangle$
5: $\langle \vec{v}_{i+1} \rangle = (1-\eta_i) \odot \langle \vec{w}_{i+1} \rangle \oplus \eta_i \odot \langle \vec{w}_i \rangle$
6: **return** $\langle \vec{w}_{i+1} \rangle$ and $\langle \vec{v}_{i+1} \rangle$

5.1 Depth Reduction

We now show how to reduce the depth to four by removing a dependency and duplicating some work. Specifically, we remove the dependency in Algorithm 6 of $\langle \vec{v}_{i+1} \rangle$ on $\langle \vec{w}_{i+1} \rangle$ by observing:

$$\begin{aligned}\langle \vec{v}_{i+1} \rangle &= (1-\eta_i) \odot \langle \vec{w}_{i+1} \rangle \oplus \eta_i \odot \langle \vec{w}_i \rangle \\ &= (1-\eta_i) \odot \langle \vec{v}_i \rangle \oplus (1-\eta) \langle \vec{c}_i \rangle \oplus \eta_i \odot \langle \vec{w}_i \rangle\end{aligned}$$

Due to the linearity of SumRows, we define:

$$\begin{aligned}\langle \vec{c'}_i \rangle &= (1-\eta) \langle \vec{c}_i \rangle \\ &= \mathsf{SumRows}(((-(1-\eta)\gamma c_3 \odot \langle Z_i \rangle) \\ &\qquad \odot \langle \vec{\ell}_i^\intercal \rangle) \odot M_i \oplus ((1-\eta)\gamma c_0 \odot \langle Z_i \rangle)).\end{aligned}$$

Thus, at the cost of additional computation, we can compute $\langle \vec{c'}_i \rangle$ in parallel with $\langle \vec{c}_i \rangle$, so we need not wait for the value of $\vec{w}_{i+1}$ to compute $\vec{v}_{i+1}$. This reduces the depth of the circuit to four as shown in Figure 1, at the cost of an increase in the number of gates. However, using a parallel evaluator there need not be a corresponding increase in evaluation time.

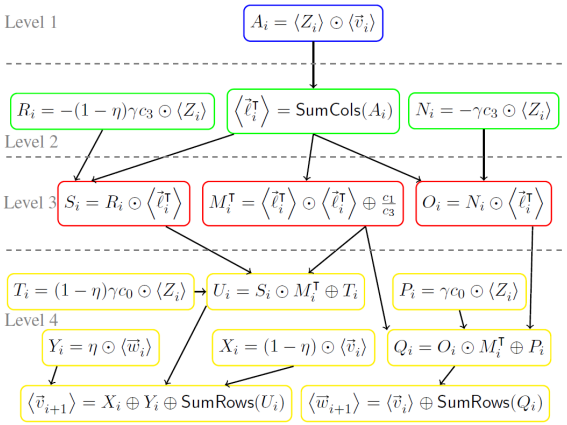

Figure 1: Depth-4 circuit for the computation of $\langle \vec{w}_{i+1} \rangle$ and $\langle \vec{v}_{i+1} \rangle$. Inputs are γ, η_i, $\langle Z_i \rangle$, $\langle \vec{w}_i \rangle$, and $\langle \vec{v}_i \rangle$. Operations at each level are grouped by color. Encoded variables correspond to Algorithm 6. All values are encoded with respect to an arbitrary $m \times n$ unit.

5.2 Use in Logistic Regression Training

The client encrypts each mini-batch Z_i to be used for training and sends them to the server along with evaluation keys. The learning rate α is a public parameter sent in-the-clear. If asymmetric CKKS encryption is used, the server can generate fresh encryptions of $\vec{0}$ for $\langle \vec{w} \rangle$ and $\langle \vec{v} \rangle$, otherwise the client must generate these ciphertexts and sends them along with the mini-batches.[6]

Encodings In [13], the mini-batch size f was always chosen to be the first dimension of the encoding unit (which must be a power of two), while the second dimension of the encoding unit depended on the exact CKKS parameters selected (i.e., the number of plaintext slots) and the number of features in the training set. Using the generalized linear algebra operations and encodings from Section 3, we can decouple the encoding

[5] Note that $-\gamma c_3 \odot \langle Z_i \rangle$ does not depend on previous lines, so it can be computed in parallel with line 1 of Algorithm 6.

[6] Using symmetric encryption also enables some ciphertext compression. It therefore may be more space efficient to use symmetric encryption, despite having to send extra ciphertexts.

unit from the mini-batch size and use an *arbitrary* encoding unit as in Algorithm 6, and an arbitrary mini-batch size f (not necessarily a power of two). One benefit is that since m must be a power of two, [13] only worked for mini-batches which contained a power of two number of training samples; our generalization has no such restriction. A second benefit is that we can divide mini-batches along both dimensions, giving even more encoding flexibility. As an example, consider training on mini-batches of size 64×128 using CKKS parameters with 4096 plaintext slots. Using the techniques of [13], we must use a 64×64 encoding unit, meaning each mini-batch is split between two ciphertexts, and each 128-dimensional column vector (e.g., $\vec{v}$) is also split into two ciphertexts. However, Section 3 allows us to use a 32×128 encoding unit, which results in two ciphertexts for the minibatch, but only one ciphertext for the vector. Thus by using the generalized encoding structure, we are able to reduce the number of ciphertexts involved in the computation, which can reduce communication overhead and improve performance.

6 DEPTH 2.5 LOGISTIC REGRESSION TRAINING

In this section, we will further reduce the multiplicative depth of a training iteration using two techniques from compilers. First, we increase the parallelism of the circuit by removing the dependency of $\vec{\ell}_i$ on $\vec{v}_i$ from the previous iteration (see Algorithm 5). We can therefore treat $\vec{\ell}_i$ as an *input* to each training iteration and instead compute $\vec{\ell}_{i+1}$ in parallel with $\vec{v}_{i+1}$ and $\vec{w}_{i+1}$. As a side effect, this introduces asymmetry in the homomorphic circuit between even and odd training iterations, so we utilize *loop unrolling* to compute two training iterations in each iteration of the for loop. By carefully designing the circuit for the unrolled loop, the first level of multiplications is independent of the outputs of the previous for loop iteration. This allows us to employ *pipelining* to start the next unrolled loop iteration before completing the previous iteration. We give a depth-five circuit for computing two training iterations, yielding an average depth of 2.5 per iteration. As with the previous circuit, all homomorphic values are encoded with respect to one arbitrary encoding unit, and the circuit works for training with arbitrary-size mini-batches (i.e., not necessarily a power of two).

6.1 Adding Parallelism

We use the technique from Section 5 to substitute the definition of $\vec{v}_{i+1}$ into $\vec{\ell}_{i+1}$ to remove the dependency, allowing us to compute $\vec{\ell}_{i+1}$ in parallel with $\vec{w}_{i+1}$ and $\vec{v}_{i+1}$. Expanding out the definition of $\vec{\ell}_{i+1}$ from Algorithm 5:

$$\begin{aligned}\vec{\ell}_{i+1} &= Z_{i+1} \cdot ((1-\eta_i)(\vec{v}_i + \gamma \cdot Z_i^\intercal \cdot \overrightarrow{\sigma'}(-\vec{\ell}_i)) + \eta_i \vec{w}_i) \\ &= (1-\eta_i) \cdot Z_{i+1} \cdot \vec{v}_i + (1-\eta_i)\gamma \cdot Z_{i+1} Z_i^\intercal \cdot \overrightarrow{\sigma'}(-\vec{\ell}_i) \\ &\quad + \eta_i \cdot Z_{i+1} \cdot \vec{w}_i \end{aligned} \tag{6.1}$$

The key to reducing the multiplicative depth is to compute (or precompute) a multiple of $Z_{i+1}Z_i^\intercal$, which has no computational dependencies, rather than sequentially multiplying by the two mini-batch matrices as Figure 1. Homomorphically, our goal is to compute $\langle \vec{\ell}_{i+1}^\intercal \rangle$, since this is the value needed in the computation of $\langle \vec{v}_{i+2} \rangle$ and $\langle \vec{w}_{i+2} \rangle$ (cf. Figure 1).

The homomorphic computation, however, apparently results in a paradox: $\vec{v}_i$ is encoded as a column vector, so the result of homomorphically computing the first term in Equation (6.1) is an encoded row vector. On the other hand, given $\langle \vec{\ell}_i^\intercal \rangle$ as in previous circuits, the middle term is the product of a (single) matrix and a row vector, which results in a column vector; we cannot add these two encoded vectors.

To solve this problem, we will instead homomorphically compute $\langle \vec{\ell}_{i+1}^\intercal \rangle$ by taking the transpose of (only) the middle term in Equation (6.1), to get $(1-\eta_i)\gamma \cdot \overrightarrow{\sigma'}(-\vec{\ell}_i^\intercal) \cdot Z_i Z_{i+1}^\intercal$. Note that mathematically, transposing this term corresponds to mixing row and column vectors, but homomorphically, each term will be an (encoded) row vector. The middle term is similar to the homomorphic computation of $\langle \vec{c}_i \rangle$ from Algorithm 6, except that we need to compute $\sigma'(\bullet)$ on the row vector $\vec{\ell}_i^\intercal$, and we use the square matrix $Z_i Z_{i+1}^\intercal$ rather than Z_i. If we treat the matrix product as an input, we can transpose each piece of the computation of $\langle \vec{c}_i \rangle$ to compute this term. First, define $M_i = \langle \vec{\ell}_i \rangle \odot \langle \vec{\ell}_i \rangle \oplus \frac{c_1}{c_3}$, then:

$$\begin{aligned}&\left\langle (1-\eta_i)\gamma \cdot \overrightarrow{\sigma'}(-\vec{\ell}_i^\intercal) \cdot Z_i Z_{i+1}^\intercal \right\rangle \\ &= \mathsf{SumCols}(((-(1-\eta_i)\gamma c_3 \odot \langle Z_{i+1} Z_i^\intercal \rangle) \\ &\qquad \odot \langle \vec{\ell}_i \rangle) \odot M_i \oplus ((1-\eta_i)\gamma c_0 \odot \langle Z_{i+1} Z_i^\intercal \rangle))\end{aligned}$$

Computing $\langle \vec{\ell}_{i+1}^\intercal \rangle$ We now return to the problem of computing $\langle \vec{\ell}_{i+1}^\intercal \rangle$. We have seen how to compute the middle term, so we just need to add the outer terms of Equation (6.1):

$$\begin{aligned}\langle \vec{\ell}_{i+1}^\intercal \rangle = \; & (1-\eta_i) \odot \mathsf{SumCols}(\langle Z_{i+1} \rangle \odot \langle \vec{v}_i \rangle) \\ \oplus \; & \mathsf{SumCols}(((-(1-\eta_i)\gamma c_3 \odot \langle Z_{i+1} Z_i^\intercal \rangle) \odot \langle \vec{\ell}_i \rangle) \odot M_i \\ & \qquad \oplus ((1-\eta_i)\gamma c_0 \odot \langle Z_{i+1} Z_i^\intercal \rangle)) \\ \oplus \; & \eta_i \odot \mathsf{SumCols}(\langle Z_{i+1} \rangle \odot \langle \vec{w}_i \rangle)\end{aligned}$$

For now, we assume that $F_i = \langle -(1-\eta_i)\gamma c_3 Z_{i+1} Z_i \rangle$ is available as an input to this computation; we explore this assumption in Section 6.5. Overall, this leads to a depth three circuit for computing $\langle \vec{\ell}_{i+1}^\intercal \rangle$, given in Figure 2.

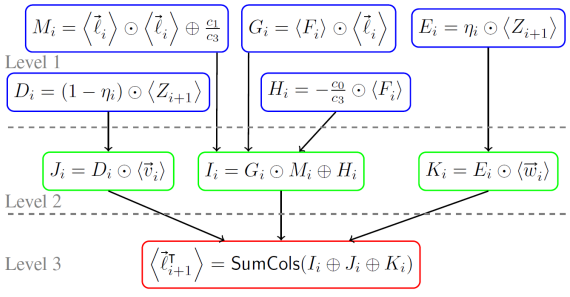

Figure 2: Depth-3 circuit for the computation of $\langle \vec{\ell}_{i+1}^\intercal \rangle$. Inputs are $\langle F_i \rangle$, $\langle Z_{i+1} \rangle$, $\langle \vec{w}_i \rangle$, $\langle \vec{v}_i \rangle$, and $\langle \vec{\ell}_i \rangle$. Operations at each level are grouped by color. Note the use of the extended SumCols linear map since the dimensions of I and J do not match. All values are encoded with respect to an arbitrary encoding unit.

Computing $\langle \vec{v}_{i+1} \rangle$ and $\langle \vec{w}_{i+1} \rangle$ Treating $\langle \vec{\ell}_i^\intercal \rangle$ as an input, we can modify Figure 1 to compute $\langle \vec{v}_{i+1} \rangle$ and $\langle \vec{w}_{i+1} \rangle$ in depth three, simply by removing the temporary value $\langle A_i \rangle$. This circuit is shown in Figure 3.

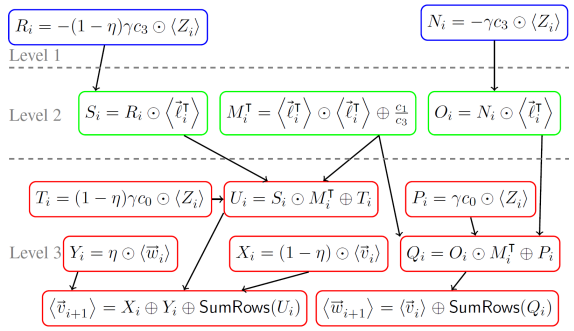

Figure 3: Depth-three circuit layout for computing $\langle \vec{w}_{i+1} \rangle^{[m \times n]}$ and $\langle \vec{v}_{i+1} \rangle^{[m \times n]}$. Inputs are $\langle Z_i \rangle^{[m \times n]}$, $\langle \vec{\ell}_i^\intercal \rangle^{[m \times n]}$, $\langle \vec{w}_i \rangle^{[m \times n]}$, and $\langle \vec{v}_i \rangle^{[m \times n]}$. Operations at each level are grouped by color.

Computing Dependencies In the process of creating a low-depth circuit for $\langle \vec{\ell}_{i+1}^\intercal \rangle$ (Figure 2), we have introduced additional dependencies. At first glance, Figure 2 requires $\langle \vec{\ell}_i \rangle$, which can be computed by taking the transpose of Figure 2; see Figure 4. However, the transpose circuit requires $\langle \vec{w}^\intercal \rangle$ and $\langle \vec{v}^\intercal \rangle$, which are computed in Figure 5. This essentially doubles our circuit size: rather than just computing $\langle \vec{v} \rangle$, $\langle \vec{w} \rangle$, and $\langle \vec{\ell} \rangle$ in each iteration, we must also compute their transposes, which increases the amount to of computation required. Table 1 provides a summary of these four circuit components.

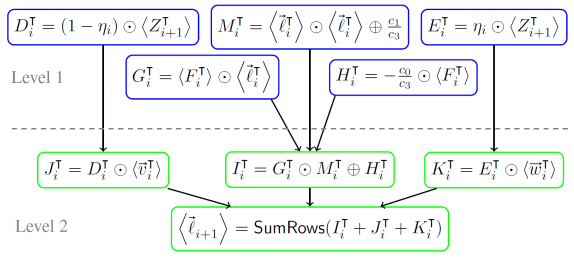

Figure 4: Depth-2 circuit layout for the computation of $\langle \vec{\ell}_{i+1} \rangle^{[n \times m]}$. Inputs are $\langle F_i^\intercal \rangle^{[m \times n]}$, $\langle Z_{i+1}^\intercal \rangle^{[n \times m]}$, $\langle \vec{w}_i^\intercal \rangle^{[n \times m]}$, $\langle \vec{v}_i^\intercal \rangle^{[n \times m]}$, and $\langle \vec{\ell}_i^\intercal \rangle^{[m \times n]}$. Operations at each level are grouped by color. Note the use of SumRows as an additive homomorphism.

6.2 Loop Unrolling

Unfortunately, the components in Table 1 are misaligned in terms of multiplicative depth. In isolation, these circuits can be run in parallel, but the overall depth is still four per iteration. By unrolling the loop to look at two consecutive iterations, we can achieve better alignment and lower depth. Appendix A shows how to align the four figures in the table so that all required inputs are computed in time. By running these four

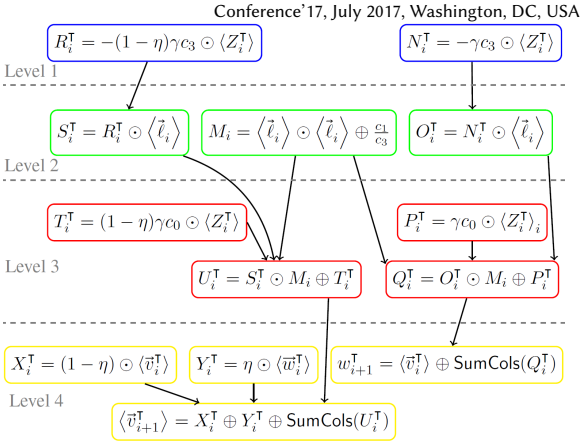

Figure 5: Depth-four circuit for computing $\langle \vec{w}_{i+1}^\intercal \rangle^{[n \times m]}$ and $\langle \vec{v}_{i+1}^\intercal \rangle^{[n \times m]}$. Inputs are $\langle Z_i^\intercal \rangle^{[n \times m]}$, $\langle \vec{\ell}_i \rangle^{[n \times m]}$, $\langle \vec{w}_i^\intercal \rangle^{[n \times m]}$, and $\langle \vec{v}_i^\intercal \rangle^{[n \times m]}$. Operations at each level are grouped by color.

Circuit	Depth	Inputs	Outputs
Figure 2	3	$\langle F_i^\intercal \rangle, \langle \vec{\ell}_i \rangle, \langle Z_{i+1} \rangle, \langle \vec{w}_i^\intercal \rangle, \langle \vec{v}_i^\intercal \rangle$	$\langle \vec{\ell}_{i+1}^\intercal \rangle$
Figure 3	3	$\langle \vec{w}_i \rangle, \langle \vec{v}_i \rangle, \langle \vec{\ell}_i^\intercal \rangle, \langle Z_i \rangle$	$\langle \vec{w}_{i+1} \rangle, \langle \vec{v}_{i+1} \rangle$
Figure 4	2	$\langle F_i \rangle, \langle \vec{\ell}_i^\intercal \rangle, \langle Z_{i+1}^\intercal \rangle, \langle \vec{w}_i \rangle, \langle \vec{v}_i \rangle$	$\langle \vec{\ell}_{i+1} \rangle$
Figure 5	4	$\langle \vec{w}_i^\intercal \rangle, \langle \vec{v}_i^\intercal \rangle, \langle \vec{\ell}_i \rangle, \langle Z_i^\intercal \rangle$	$\langle \vec{w}_{i+1}^\intercal \rangle, \langle \vec{v}_{i+1}^\intercal \rangle$

Table 1: Low-depth circuits for individual components of the logistic regression training circuit.

circuits in parallel[7] (the "full" circuit), we can compute two consecutive iterations of the training loop with depth six, giving an average of depth three per training iteration.

6.3 Pipelining

Notice that the Figures 7, 8, and 10 have depth five, while the first computation level in Figure 9 (depth six) involves only computing scalar multiples of $Z_i^\intercal$. By pipelining multiple iterations of the training algorithm, we can start evaluating the circuit for the next two training iterations prior to finishing the evaluation of the circuit for the previous two training iterations, bringing the depth for two iterations to five. The values computed in the first level of the full circuit are $R_i^\intercal := -\gamma c_3 (1-\eta_i) \odot \langle Z_i^\intercal \rangle$ and $N_i^\intercal := -\gamma c_3 \odot \langle Z_i^\intercal \rangle$.

6.4 Using Mixed Encoding Units for Compactness

The "uniform-unit" circuit described above the same encoding unit for a value *and* its transpose. As a result, it is not optimal in terms of the number of ciphertexts used to encode

[7] Note that a naïve implementation results in duplicate computation for some values; an efficient implementation would overlap these four circuits to eliminate duplication.

objects and total number of operations performed. For example, in [13], the row vector $\vec{\ell}_i^\intercal \in \mathbb{R}^m$ is encoded as the columns of an $m \times n$ encoding unit. The circuit described above *also* uses this encoding unit for $\vec{\ell}_i$, which is encoded as the rows of an $m \times n$ unit. This may result in padding or multiple ciphertexts depending on the number of training features. We can achieve a more compact encoding by using *different* encoding units for certain values. Due to space constraints, we leave the description of this variant to the full version of the paper.

6.5 Use in Logistic Regression Training

In [13] and Section 5.1, clients must encrypt each mini-batch Z_i. However, the low-depth circuit described in this section requires additional inputs. In particular, the first level of Figures 7—10 require F_0, N_0, R_0, and their transposes, and $Z_i^\intercal$ are needed throughout the computation. There are many possible tradeoffs between which of these values are computed by the client, and which are computed by the server. We describe two of these options below.

Client Computation Circuit In this version of the circuit, the client encrypts Z_i, $Z_i^\intercal$, the matrix products F_i, $F_i^\intercal$ and, in the case of symmetric encryption, the all-0 vectors $\vec{w}_0$, $\vec{w}_0^\intercal$, $\vec{v}_0$, $\vec{v}_0^\intercal$, $\vec{\ell}_0$, and $\vec{\ell}_0^\intercal$. The server computes the initial loop variables N_0, $N_0^\intercal$, R_0, and $R_0^\intercal$. With this version, the client sends about $4k$ encrypted inputs[8] for k iterations. Nominally, this results in a circuit where the first iteration of Figures 7—10 have depth six (since the server must compute $R_0^\intercal$, etc.), while subsequent iterations have depth five (by applying pipelining), giving an overall circuit depth of $2.5k + 1$ for k iterations.[9]

Server Computation Circuit As an alternative, we can use Algorithm 3 to have the server compute most of the F_i and $F_i^\intercal$ (not depicted in Figures 7—10). This dramatically increases the server computation, while simultaneously reducing the client computation and communication overhead. Concretely, the client encrypts Z_i, $Z_i^\intercal$, F_0, $F_0^\intercal$, and, in the case of symmetric encryption, the all-0 vectors $\vec{w}_0$, $\vec{w}_0^\intercal$, $\vec{v}_0$, $\vec{v}_0^\intercal$, $\vec{\ell}_0$, and $\vec{\ell}_0^\intercal$. Again, this circuit has depth $2.5k+1$ for k iterations. As the number of training iterations increases, the amount of client computation saved increases, and the communication savings increase as well. As long as there is sufficient parallelism available on the evaluator, this approach offers reduced client computation and communication at no runtime cost.

Encodings All inputs and intermediate values can be encoded with a single arbitrary encoding unit. However, this may not lead to the most compact representation of objects. For example, in [13], the row vector $\vec{\ell}_i^\intercal \in \mathbb{R}^m$ is encoded as the columns of an $m \times n$ encoding unit. If we use a single unit in our circuit, then we would also encode $\vec{\ell}_i$ as the rows of an $m \times n$ unit, which may result in padding and wasted space, though this doesn't affect circuit depth. A more compact representation could be achieved by encoding $\vec{\ell}_i$ as the rows of an $n \times m$ unit; see [12] for details.

[8]There may be more than $4k$ *ciphertexts*, depending on the encoding unit and the number of features.

[9]By carefully arranging the first few levels of either version of the circuit described here, we can save one level and create a circuit with depth exactly $2.5 \cdot k$ for k iterations on average.

7 EVALUATION

We implemented the circuit from [13] and both variations of the Section 6 circuit with the SEAL homomorphic encryption library [20] and circuit-level parallelism. Encryption parameters were selected to conform to the 128-bit security level as standardized in [1], or were extrapolated from this standard for larger rings at the 128-bit target security level. We tested our circuits by training a logistic regression model on the MNIST data set [17]. This data set consists of images of hand-written digits. For our binary classification problem, we restricted the training set and test set to images of the digits 3 and 8. After downsampling, each image is 14x14 pixels, each of which becomes a feature in the model. We evaluated the performance on an Amazon EC2 c5.24xlarge instance, which has 96 CPU threads.

In practice, an implementation of encrypted logistic regression training would fix the number of training iterations performed by a single circuit and then use bootstrapping as in [13] to achieve a larger number of iterations. Our implementation does not include bootstrapping, but we emphasize that our circuits are drop-in replacements for the circuit from [13], and therefore their bootstrapping technique is trivially applicable to this work. Our performance improvements in the leveled-HE setting also carry over to the bootstrapped setting. We therefore evaluate the circuits for a fixed number of iterations and consider the following properties:

- Encryptions. Number of ciphertexts (not linear algebra objects) encrypted by the client.
- Encrypted Input Size. Total size of ciphertexts send to server. While a naïve implementation would encrypt each ciphertext at the maximum possible level and let the server reduce the levels when necessary, this unnecessarily wastes bandwidth and server computation. Our implementation instead encrypts inputs at the first level where they are needed, meaning each input has a different size, and the total input size is a weighted sum where the weights are the HE levels of the inputs. We also note that we used a public-key variant of CKKS encryption; using a symmetric-key variant can result in more compact ciphertexts resulting in a significant reduction in bandwidth.
- Circuit Depth. Multiplicative depth of the circuit.
- Circuit Width. We define the *multiplicative width* of a circuit to be the total number of multiplications performed in a circuit evaluation divided by the multiplicative depth of the circuit. Roughly, this is an indicator of the multiplications that can be performed in parallel.
- Runtime. Runtime of the circuit evaluation. This only includes server-side operations.

Table 2 compares the Section 6 circuits to the circuit from [13].

Multiplicative Depth Comparing the runtime of the low-depth circuit that computes F_i on the client to the [13] circuit

Circuit	Encryptions	Input Size (MB)	Depth	Width			Runtime (s)		
				32	64	128	32	64	128
[13]	8	177	30	———	1.8	———	———	79.2	———
This work: F_i on Client	33	593	15	———	14.9	———	———	13.1	———
This work: F_i on Server	28	386	15	89.5	164.2	313.5	70.2	137.4	273.6

Table 2: Comparison of logistic regression training circuits for six training iterations with an encoding unit of 128×256 for mini-batches of size 32, 64, and 128. Each configuration use CKKS parameters with 2^{15} plaintext slots

shows that cutting multiplicative depth in half results in even greater benefits at runtime.

Client-side vs Server-side Computation Computing matrix products on the server incurs extra computational cost during evaluation. The table shows that this cost is affected by the mini-batch size, which is not true when the client computes the matrix products. This is because Algorithm 3 performs operations based on the size of the mini-batches directly, whereas all other operations perform operations based on the size of the *encoded* mini-batches.

Parameters Affecting Performance Given parameters for the logistic regression model (e.g., mini-batch size and number of training iterations), there are still several circuit parameters which can affect performance, including number of plaintext slots and the encoding unit. Doubling the number of plaintext slots doubles the size of ciphertexts and makes all operations slower, but can give additional flexibility in selecting the encoding unit with the compact circuit. The circuits are so large (even for six iterations) it is difficult to predict how the encoding unit affects performance. We leave this as future work.

Model Accuracy Since all of the circuits compared in Table 2 evaluate the same algorithm, they should produce identical models. The accuracy of the model produced by the training algorithm is affected by the accuracy and validity of the sigmoid approximation and the noise introduced by homomorphic computation. Figure 6 shows that neither of these potential sources of error have much affect on model accuracy.

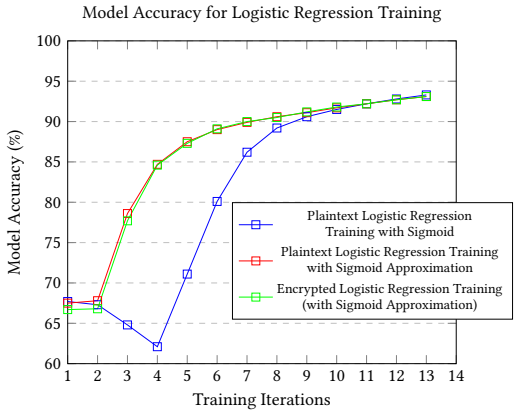

Figure 6: Average model accuracy (over 1000 models) of a logistic regression model trained on the MNIST data set. Model accuracy is computed as basic *classification accuracy*: the percentage of correctly predicted test samples.

REFERENCES

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[2] Yoshinori Aono, Takuya Hayashi, Le Trieu Phong, and Lihua Wang. 2016. Scalable and Secure Logistic Regression via Homomorphic Encryption. Cryptology ePrint Archive, Report 2016/111. https://eprint.iacr.org/2016/111.

[3] David W. Archer, Jose Manuel Calderon Trilla, Jason Dagit, Alex J. Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. Cryptology ePrint Archive, Report 2019/988. https://eprint.iacr.org/2019/988.

[4] Oluwafadekemi Areo and Obindah Gershon. 2020. Personal Income Tax Compliance in Nigeria: A Generalised Ordered Logistic Regression. *Research in World Economy* 11 (06 2020), 261. https://doi.org/10.5430/rwe.v11n3p261

[5] Saikat Banerjee, Lingyao Zeng, Heribert Schunkert, and Johannes Söding. 2018. Bayesian multiple logistic regression for case-control GWAS. *PLOS Genetics* 14 (12 2018), e1007856. https://doi.org/10.1371/journal.pgen.1007856

[6] Christine Bolton. 2010. Logistic regression and its application in credit scoring.

[7] Charlotte Bonte, Carl Bootland, Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. 2017. Faster Homomorphic Function Evaluation using Non-Integral Base Encoding. Cryptology ePrint Archive, Report 2017/333. https://eprint.iacr.org/2017/333.

[8] Charlotte Bonte and Frederik Vercauteren. 2018. Privacy-Preserving Logistic Regression Training. Cryptology ePrint Archive, Report 2018/233. https://eprint.iacr.org/2018/233.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2016. Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Report 2016/421. https://eprint.iacr.org/2016/421.

[10] Jagpreet Chhatwal, Oguzhan Alagoz, Mary Lindstrom, Charles Kahn, Jr, Katherine Shaffer, and Elizabeth Burnside. 2009. A Logistic Regression Model Based on the National Mammography Database Format to Aid Breast Cancer Diagnosis. *AJR. American journal of roentgenology* 192 (05 2009), 1117–27. https://doi.org/10.2214/AJR.07.3345

[11] Jack L.H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. Cryptology ePrint Archive, Report 2018/202. https://eprint.iacr.org/2018/202.

[12] Eric Crockett. 2020. A Low-Depth Homomorphic Circuit for Logistic Regression Model Training. Cryptology ePrint Archive, Report 2020/1483. https://eprint.iacr.org/2020/1483.

[13] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2018. Efficient Logistic Regression on Large Encrypted Data. Cryptology ePrint Archive, Report 2018/662. https://eprint.iacr.org/2018/662.

[14] HIT 2020. AWS HIT. https://github.com/awslabs/homomorphic-implementors-toolkit. Amazon Web Services.

[15] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. 2018. Logistic Regression Model Training based on the Approximate Homomorphic Encryption. Cryptology ePrint Archive, Report 2018/254. https://eprint.iacr.org/2018/254.

[16] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. 2018. Secure Logistic Regression Based on Homomorphic Encryption: Design and Evaluation. Cryptology ePrint Archive, Report 2018/074. https://eprint.iacr.org/2018/074.

[17] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

[18] Y. E. Nesterov. 1983. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Dokl. Akad. Nauk SSSR* 269 (1983),

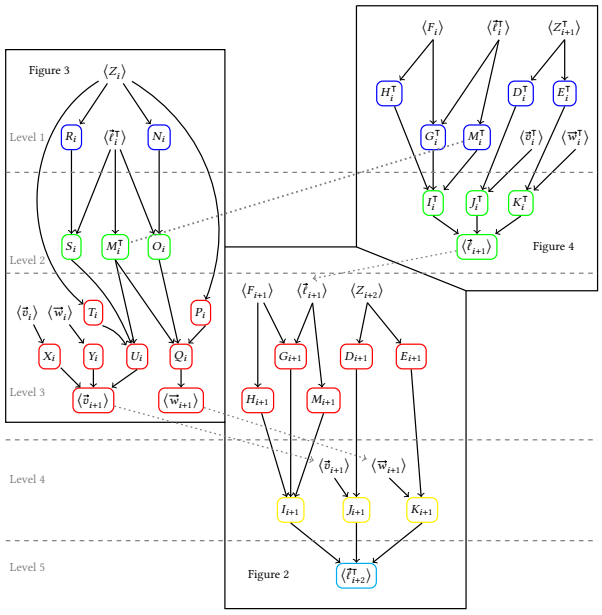

Figure 8: Unrolled computation of $\langle \vec{\ell}^{\intercal} \rangle$.

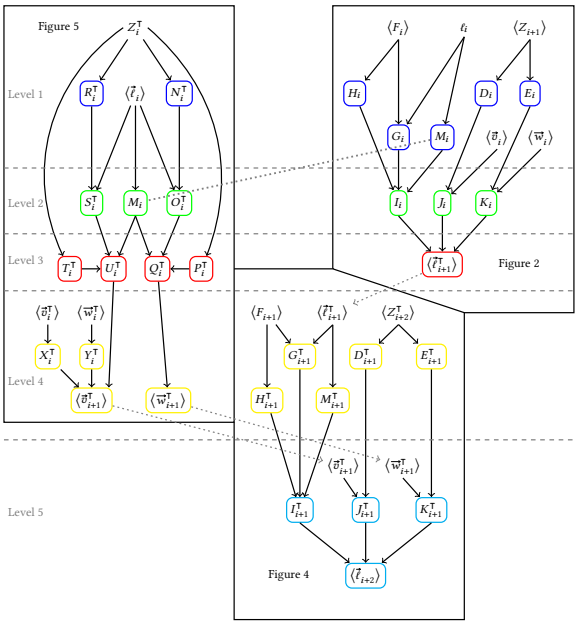

Figure 10: Unrolled computation of $\langle \vec{\ell} \rangle$.

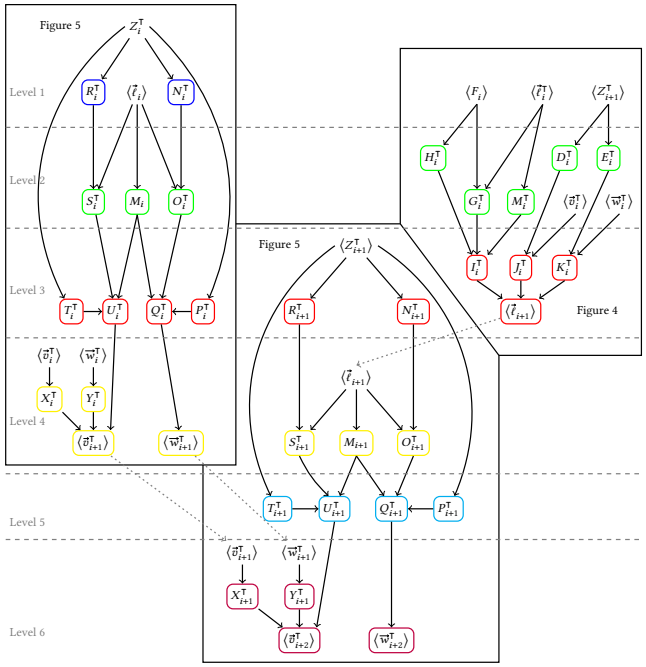

Figure 9: Unrolled computation of $\langle \vec{v}^{\intercal} \rangle$ and $\langle \vec{w}^{\intercal} \rangle$.

543–547. http://mpawankumar.info/teaching/cdt-big-data/nesterov83.pdf
[19] Tomasz Rymarczyk, Edward Kozłowski, Grzegorz Kłosowski, and Konrad Niderla. 2019. Logistic Regression for Machine Learning in Process Tomography. *Sensors* 19 (08 2019), 3400. https://doi.org/10.3390/s19153400
[20] SEAL 2020. Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

A CIRCUIT DIAGRAMS FOR LOW-DEPTH LOGISTIC REGRESSION TRAINING

This section shows how to align Figures 2—5 into a full circuit for logistic regression training. The upper panels of Figures 7—10 overlap (e.g., the top left panels of Figure 7 and Figure 8) to obtain a depth six circuit for two training iterations, which can be reduced to depth five with pipelining. Recall that we define $F_i := -\gamma c_3(1-\eta_i) \cdot Z_{i+1} Z_i^{\intercal}$. Figures have computations at the same HE level grouped by color.

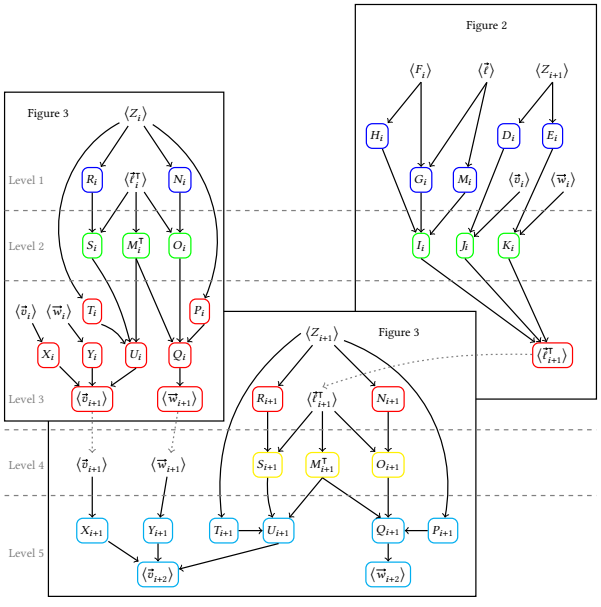

Figure 7: Unrolled computation of $\langle \vec{v} \rangle$ and $\langle \vec{w} \rangle$.