Secure Image Classification with Lattice-Based Fully Homomorphic Encryption

Sanath Govindarajan, Walden Yan
Mentor: William Moses
Scenario

You are a doctor who has a patient’s x-ray images. You want to send it to a third-party service which specializes in detecting bone defects. But you cannot legally send the images without compromising the privacy of the patient, due to HIPAA / other privacy laws. Is there no hope?
How do we perform computations without giving away the data?
Solution

We created a fully homomorphic encryption scheme that is capable of performing the operations required by a neural network. Then, we created methods of adapting and training neural networks to run efficiently with this encryption scheme, with minimal loss to accuracy.
Introducing Fully Homomorphic Encryption (FHE)[1]

- Supports arbitrary computation on encrypted data

[1] Acar, Abbas, Aksu, Hidayet, Uluagac, Selcuk, Conti, Mauro 2017
Uses of FHE

- Using FHE, we can send off our tasks to someone with a more powerful computer or a better algorithm, without worrying about data leaks.
  - Email filtering
  - Medical applications, e.g. image classification
  - Defense
  - Finance
Road Map

- We need to be able to:
  - Encrypt data
  - Homomorphically compute on the encrypted data
  - Run inference on the encrypted data using a neural network
Neural Networks
Amazing Achievements

- Translation between languages [1]
- WaveNet: speech generation [2, 3]
- Lip reading [4]
- Visual reasoning [5]

[5] Santoro, Raposo, Barrett, Malinowski, Pascanu, Battaglia, Lillicrap 2017
Neural Networks

- Mimic the brain
- Layers of artificial neurons

- Input Layer
  - Receives data for user

- Output Layer
  - Gives specific information about input

- Hidden Layers
  - Computations needed to transform input into output
The Basic Neuron
Building and Using Networks (On Images)

- **Training**
  - Requires a huge dataset of training images
  - Mathematically modify weights to fit training examples
  - Takes up a lot of time

- **Inference**
  - Feed brand new images with the correct output unknown
  - Returns what the network believes the images to be
  - Much faster than training
Required Operations

- Matrix (Tensor) addition, subtraction and multiplication
  - Division by constants

- Activation functions
  - ReLU

- Pooling
  - Max Pooling

- Generally third parties have to do these computations
  - Services cannot always be trusted to not steal information
Who Needs Trust?
Lattice-Based Cryptography: a Somewhat Homomorphic Scheme [1]

Lattice-based cryptography is based on the Learning with Errors problem

\[
\begin{pmatrix}
a & b & c \\
d & e & f \\
g & h & j \\
\end{pmatrix} \times \begin{pmatrix}
u \\
v \\
w \\
\end{pmatrix} = \begin{pmatrix}
3 \times u \\
3 \times v \\
3 \times w \\
\end{pmatrix} + \begin{pmatrix}
x \\
y \\
z \\
\end{pmatrix}
\]

Cipher  Secret Key  Plaintext ($\mu$) = 3  Error vector with small coefficients

Constraints of FHE

- Gentry’s Scheme makes multiplication and addition homomorphic in practice, but with the restriction that $\mu \in \{0, 1\}$
  - Use list of encrypted binary digits to represent larger numbers
  - Use fixed-point numbers instead of floating-point numbers for practicality and security
- NOT and NAND are both homomorphic (and NAND is functionally complete)
  - $\text{NOT}(A) = 1 - A$
  - $\text{NAND}(A, B) = 1 - AB$
- Besides NAND and NOT, we also implemented optimized versions of AND, OR, and XOR
Useful Primitives for Machine Learning

- Multiplication and addition allow us to approximate any function by Taylor expansion (e.g. softmax, sigmoid)
- Binary adders and multipliers can be built from bit-wise operations (e.g. half-adder)
- We can also use Newton’s Method to approximate the $n^{th}$ root, which in turn can be used in the $L^n$ norm, an approximation for the max function, used in max pooling.
- ReLU in particular can be highly optimized in this scheme

<table>
<thead>
<tr>
<th></th>
<th>DONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract</td>
<td></td>
</tr>
<tr>
<td>Multiplication</td>
<td></td>
</tr>
<tr>
<td>Scalar Division</td>
<td></td>
</tr>
<tr>
<td>ReLU</td>
<td></td>
</tr>
<tr>
<td>Max Pool</td>
<td></td>
</tr>
</tbody>
</table>
Example Optimization - ReLU

ReLU($x$) = \frac{x + \lfloor |x| \rfloor}{2} requires the very costly operations of addition and division. However, our optimized version of ReLU($x$) requires only one efficient operation to be performed and is even more efficient than just $|x|$

We are using a ones-complement scheme. Let $x_1, x_2, ..., x_n$ be the bits of $x$, with $x_1$ being the sign bit. Let $r_1, r_2, ..., r_n$ be the bits of the result.

- Slow ReLU: as described above, \( \frac{x + \lfloor |x| \rfloor}{2} \)
- Fast ReLU: \( r_i = \text{OR}(x_1, x_i) \) for all \( 1 \leq i \leq n \)

<table>
<thead>
<tr>
<th></th>
<th>Fast ReLU</th>
<th>Slow ReLU</th>
<th>Abs</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>0.206</td>
<td>53.389</td>
<td>0.495</td>
<td>1.509</td>
<td>51.384</td>
</tr>
<tr>
<td>Time (relative to Fast ReLU)</td>
<td>1.0</td>
<td>259.2</td>
<td>2.4</td>
<td>7.3</td>
<td>249.4</td>
</tr>
</tbody>
</table>
Limitations of Bit List Operations

- “Traditional division” is non-trivial because long division requires a check of whether the divisor is larger than the dividend at each step.
  - We can still do “division” by representing each number by a numerator and a denominator and multiplying by the reciprocal: \( \frac{10}{1} ÷ \frac{5}{2} = \frac{(10)(2)}{(1)(5)} = \frac{20}{5} \)
  - We cannot account for any overflows because we have no information about the cipher’s bits. Especially significant because fixed-point arithmetic is used.

- Computations on bit lists with more precision are significantly slower than those with less precision
Putting It Together
Inference With Neural Networks

User

Network

YES/NO
Training vs Inference

Training
- Networks owned by other parties use unencrypted weights
- Trained with unencrypted values

Inference
- Done with encrypted values
- Not a problem if encryption makes the network slightly slower
Need of Changes

- Tempting to just encrypt everything with the FHE scheme
- The network would run too slowly, even for inference
- Make some optimizations to use less bits
  - Less bits means faster computations
Representation of Values

- $N$-bit precision encrypted integer tensors accompanied by full-precision scalars

- Input
  - Fit to $[-1, 1]$
  - Scalar of $\frac{1}{2^{N-1}-1}$

- Weights
  - Multiple previous work on weight compression [1, 2]
  - We use a combination of their methods

$$0.5743 \begin{bmatrix} 9 & -13 & 5 & 2 \\ 1 & 11 & 7 & -6 \\ 3 & -7 & -4 & 1 \\ 6 & 0 & 7 & 10 \end{bmatrix}$$

[1] Leng, Duo, Li, Zhu, Jin 2017
Quantization of ResNet

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>72.6%</td>
</tr>
<tr>
<td>3</td>
<td>72.99%</td>
</tr>
<tr>
<td>4</td>
<td>73.64%</td>
</tr>
<tr>
<td>5</td>
<td>73.71%</td>
</tr>
<tr>
<td>no quantization</td>
<td>74.28%</td>
</tr>
</tbody>
</table>

Running ResNet-18 (He et al. 2015) on CIFAR-100

- Little loss in accuracy
- Not the major source of imprecision
Training Low Precision Weights

- Existing weights cannot just be approximated
  - Have to be slightly altered through additional training
  - Still retains high accuracy [1]

- Maintain real-value weights
- Calculate low-precision weights before training
- Run the network on LP weights but only update full-precision weights

- Allows us to both adapt and train networks from scratch

[1] Zhou, Yao, Guo, Xu, Chen 2017
Operating on Low Precision Values
Addition

- First divide integer values by 2
- Multiply their scalar values by 2 so overall value stays the same
- Scale the integer values with the smaller scalar value to match that of the other tensor
- Simply add the encrypted values using the FHE scheme
Secure Scaling

- When sending values through the network, we encounter problems with overflow

- Solution: secure-scale operation

- We are able to scale potentially overflowing values into required bounds

- Able to perform without data leaks
  - Compute running maximums for the integer values during training
Computing ReLU

- ReLU can be done efficiently under the FHE scheme
- Networks with ReLU tend to have large negative values
- Leads to unused bits after ReLU operation
- Perfect time to use a secure-scale operation
Matrix Multiplication

- Uses standard matrix multiplication

- Face issue of overflow
  - Each element of the result is the cross product of two potentially very large vectors

- Scaling would have to be done before the multiplication
  - Just secure-scaling the input creates too much loss of precision and would require more bits

- Leads us to do a “two-phase” cross product with intermediate scaling
  - Compute by blocks and then blocks are combined
  - Scaling before and after combination of blocks
Everything is Ready

- Representation of values that allows fast computations

- Weights can be compressed without loss in accuracy

- We can avoid overflow while operating on LP values
Implementation

- Tested on “inaccessible” values
  - $N$-bit integer values
  - Act like FHE values
- Tested on MNIST dataset
  - 28 x 28 pixel handwritten digits
Network Architecture

- Three fully connected layers
- Two ReLU Layers
- 8 bits of precision
# Results

<table>
<thead>
<tr>
<th>Net Type</th>
<th>Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST Net</td>
<td>98.19%</td>
</tr>
<tr>
<td>8-bit MNIST Net</td>
<td>98.00%</td>
</tr>
</tbody>
</table>

- Adapting to be able to use encrypted values results in only a 0.19% loss in accuracy.
Conclusion

- We have made a secure FHE scheme that can do all the needed operations
- We have built a working library for the FHE scheme

- Network weights can be quantized without much loss in accuracy
- Existing networks can be adapted to this quantized scheme with slight modifications

- Showed networks can perform inference on encrypted data with minimal loss of accuracy
Future Work

- Time and memory optimizations
- Experimenting with one’s vs two’s complement
- Synthesizing optimized FHE for any operation
- Further decreasing the number of bits needed to retain high accuracy
- Adapting all types of neural network layers to run with low precision
Acknowledgements

- Our Mentor, William Moses
- Our Parents
- The PRIMES Program
Supplemental Slides
Gentry solves the problem of cipher multiplication by using a set of constructions and functions which bounds the error growth to $N+1$ per operation, where $N$ is the dimension of the matrix. Not fully homomorphic, but much better than the previous bound.
Limitations of Gentry’s Scheme

- This scheme requires that $\mu \in \{0, 1\}$
- The error still grows, just not nearly as much as before. So the modulus must be very large to account for this.
- It was proved[source] that we can build up any function by stringing together a number of NAND gates, so all we need is a homomorphic NAND.
If we have two one-bit ciphers A and B, then NAND(A, B) = 1 – AB. Since 1 is represented by the identity matrix, and addition, subtraction, and multiplication are homomorphic, NAND is also homomorphic. Equivalently, Gentry’s scheme is fully homomorphic.

Unfortunately, stringing together NAND gates always works in theory, but in practice, can be very inefficient and slow…
Lattice-Based Cryptography is Somewhat Homomorphic

- Somewhat Homomorphic Encryption: similar to FHE but there is a small error which increases with each operation.
- Eigenvalues add and multiply corresponding to matrix additions and multiplications, very useful for homomorphic encryption!
- Since the plaintext and secret key are not exact, they are only somewhat homomorphic instead of fully homomorphic.
  - Multiplicative: $B^{2^L}$ where $B$ is the maximum coefficient of the ciphertexts (aka some large prime modulus) and $L$ is the multiplicative depth of the circuit. Clearly this is not sustainable!
Operations on Bit Lists

- Addition: addition with carry can be done using half-adders and full-adders

- Multiplication: can be done using repeated additions and bit-shifts (bit-shifts do not compromise any information)

- Negation: ones-complement still works. As mentioned, NOT(A) = 1 – A is homomorphic.
Given weights \( W \in \mathbb{R}^m \) and \( N \) bits, we want

\[
\min_{\omega, G} \|W - \omega G\|_2^2
\]

s.t. \( G \in \{0, \pm 1, \pm 2, \ldots, \pm (2^{N-1}-1)\}^m \)

Let,

\[
\omega_0 := \frac{\max_{x \in W} |x|}{2^{N-1} - 1}
\]

Then,

\[
G_{k+1} := \prod_{\{0, \pm 1, \pm 2, \ldots, \pm (2^{N-1}-1)\}} \frac{W}{\omega_k}
\]

\[
\omega_{k+1} := \frac{W^T G_{k+1}}{G_{k+1}^T G_{k+1}}
\]

(\( \prod \) denotes the euclidean projection operator)
Matrix Addition

Given matrices $A$ and $B$, and scalars $\alpha$ and $\beta$ s.t. $\alpha > \beta$,

$$\gamma = 2\alpha$$

$$C = \left[ \frac{A}{2} \right] + \left[ \frac{\beta}{\alpha} \frac{B}{2} \right]$$

$$\alpha A + \beta B \rightarrow \gamma C$$

Note we divide by 2 to avoid overflow
"Scaling" Layers

- Meant to ensure bits are not being wasted
- Use unencrypted low precision values during training

When it receives inputs $\alpha$ and $A$ while training,

\[ M_B := \max_{x \in A} |x| \]

\[ M := \rho M + (1 - \rho)M_B \quad (0 < \rho < 1) \]

\[ \alpha := \frac{\alpha M_B}{2^{N-1} - 1} \]

\[ A := \left[ \left( \frac{2^{N-1} - 1}{M_B} \right) A \right] \]
Inference With Scaling Layers

- LP values are encrypted

When it receives inputs $\alpha$ and $A$ during inference,

$$\alpha := \frac{\alpha M}{2^{N-1} - 1}$$

$$A := \left[ \left(\frac{2^{N-1} - 1}{M}\right) A \right]$$
Two-Phase Cross Product

Given vectors $A, B \in \{0, \pm 1, \pm 2, \ldots, \pm (2^{N-1} - 1)\}^n$ and scalars $\alpha$ and $\beta$:

$$C_R := \left[\frac{A}{\sqrt{2^{N-1}} - 1}\right] \left[\frac{B}{\sqrt{2^{N-1}} - 1}\right]$$

$$\gamma_R := \alpha \beta (2^{N-1} - 1)$$

Now we need to be able to sum the elements of $\gamma_R C_R$.

This is done in two phases
Phase One

For simplicity, assume $n$ is a perfect square.

During training,

$$ M_{B_1} := \max_{0 \leq i < \sqrt{n}} \sum_{0 \leq j < \sqrt{n}} C_{R_1+j+i\sqrt{n}} $$

$$ M_1 := \rho M_1 + (1 - \rho)M_{B_1} \quad (0 < \rho < 1) $$

Training:

$$ \gamma_A := \frac{\gamma R M_{B_1}}{2^{N-1} - 1} $$

$$ C_A := \left[ \left( \frac{2^{N-1} - 1}{M_{B_1}} \right) C_R \right] $$

Inference:

$$ \gamma_A := \frac{\gamma R M_1}{2^{N-1} - 1} $$

$$ C_A := \left[ \left( \frac{2^{N-1} - 1}{M_1} \right) C_R \right] $$
Phase One Continued

Calculate $C_{B} \in \{0, \pm 1, \pm 2, ..., \pm (2^{N-1} - 1)\}^{\sqrt{n}}$ defined as

$$C_{B_i} = \sum_{0 \leq j < \sqrt{n}} C_{A_{1+j+i\sqrt{n}}}$$

$$\gamma_B := \gamma_A$$
Phase Two

During training,

\[ M_{B2} := \sum C_B \]
\[ M_2 := \rho M_2 + (1 - \rho) M_{B2} \quad (0 < \rho < 1) \]

Training:

\[ \gamma := \frac{\gamma_B M_{B2}}{2^{N-1} - 1} \]
\[ C := \sum \left[ \left( \frac{2^{N-1} - 1}{M_{B2}} \right) C_B \right] \]

Inference:

\[ \gamma := \frac{\gamma_B M_2}{2^{N-1} - 1} \]
\[ C := \left[ \left( \frac{2^{N-1} - 1}{M_2} \right) C_B \right] \]