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?

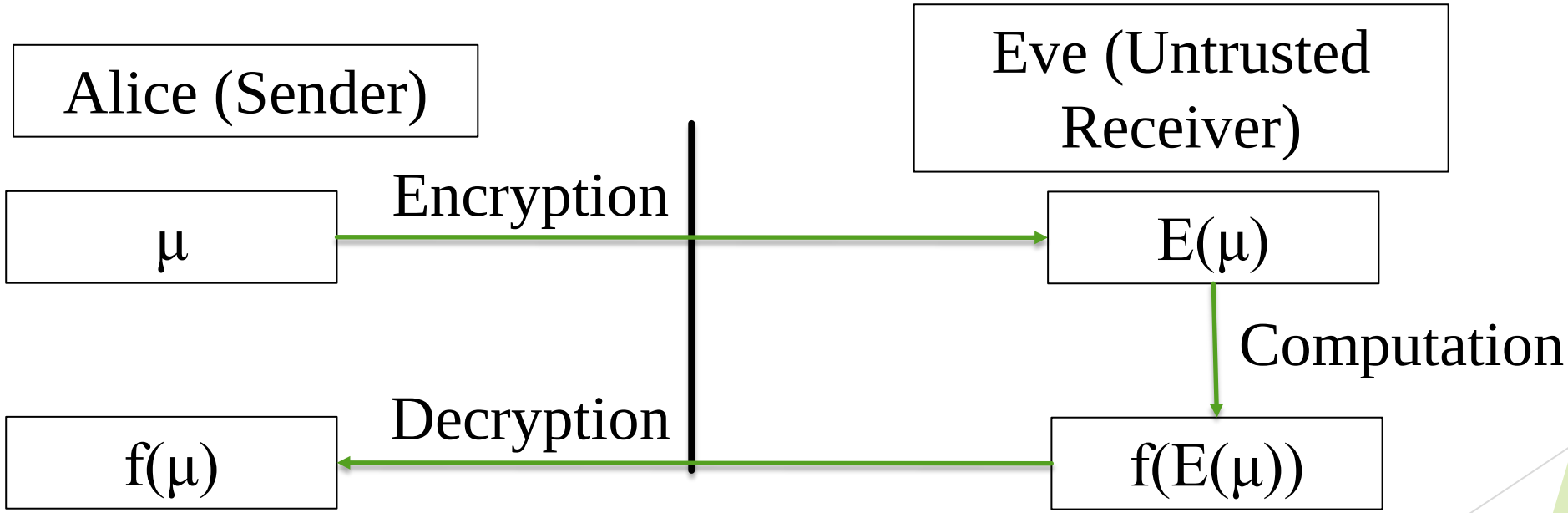
How do we perform computations without giving away the data?

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

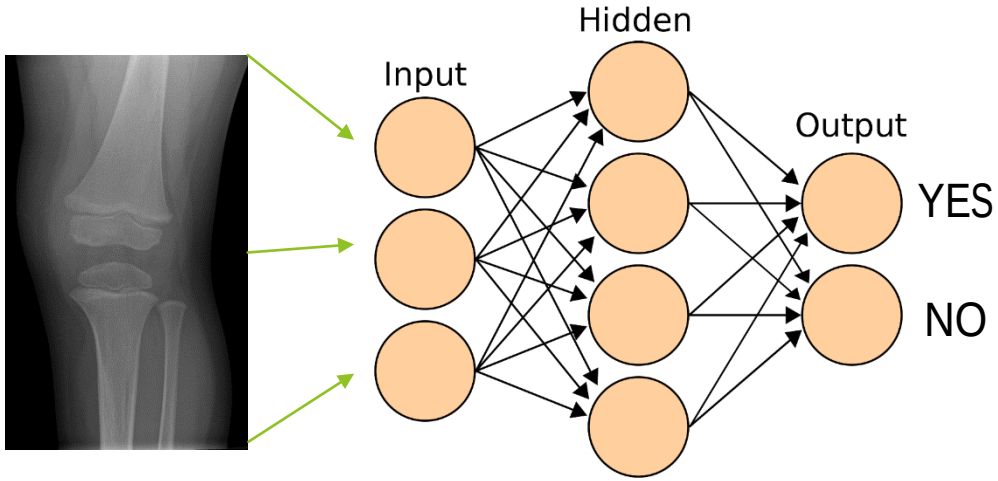
Neural Networks

Amazing Achievements

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

[1] Wu, Schuster, Chen, Le, Norouzi 2016
[2] van den Oord, Kalchbrenner, Kavukcuoglu 2016
[3] van den Oord, Kalchbrenner, Vinyals, Espeholt, Graves, Kavukcuoglu 2016
[4] Chung, Senior, Vinyals, Zisserman 2016
[5] Santoro, Raposo, Barrett, Malinowski, Pascanu, Battaglia, Lillicrap 2017

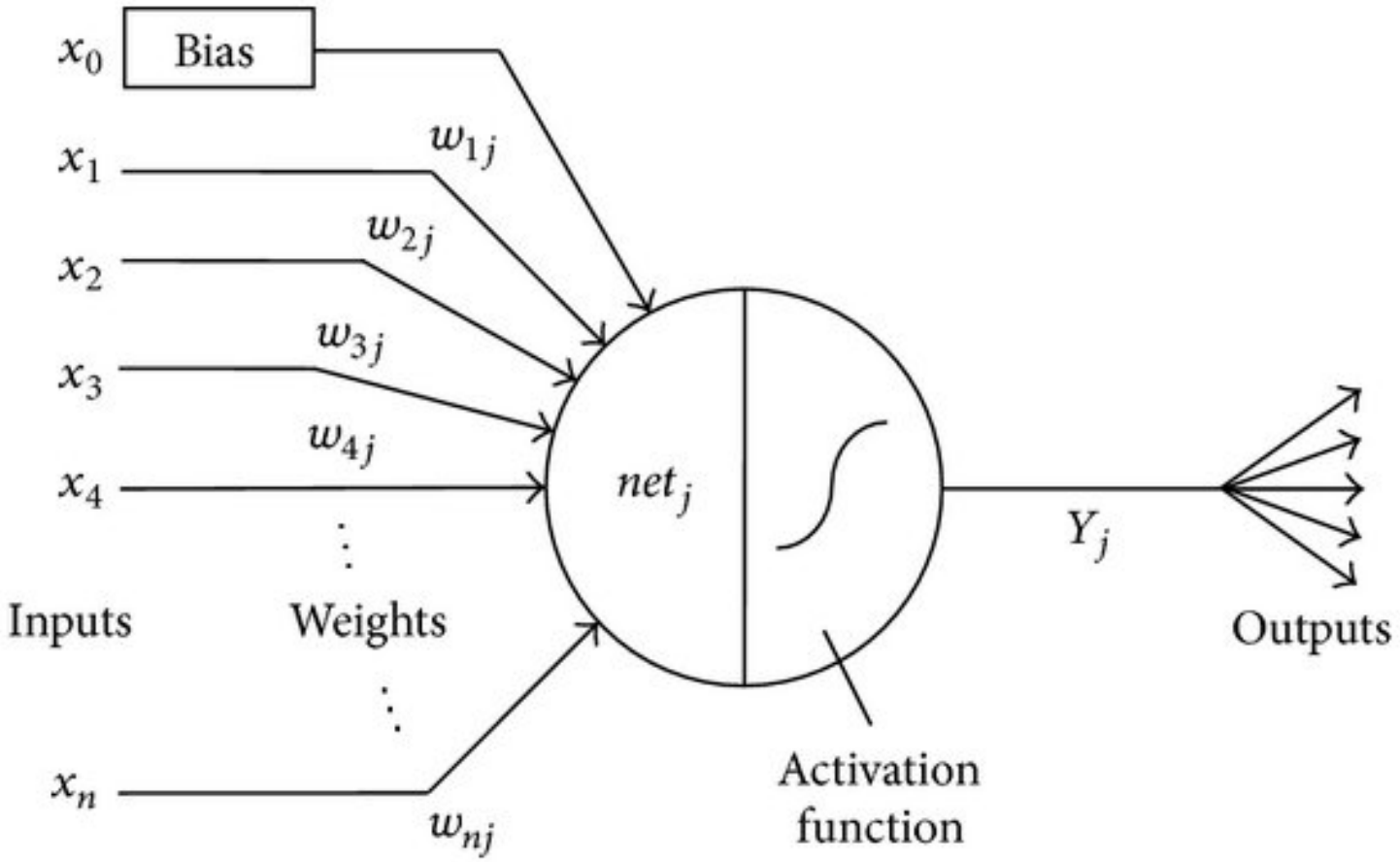
Neural Networks



- Mimic the brain
- Layers of artificial neurons

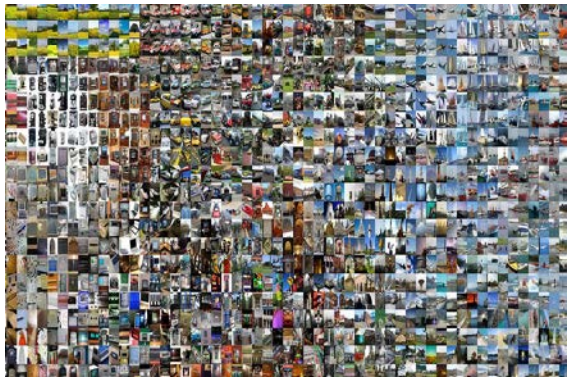
- Input Layer
  - Receives data for user
- Output Layer
  - Gives specific information about input
- Hidden Layers
  - Computations needed to transform input into output

The Basic Neuron



Building and Using Networks (On Images)

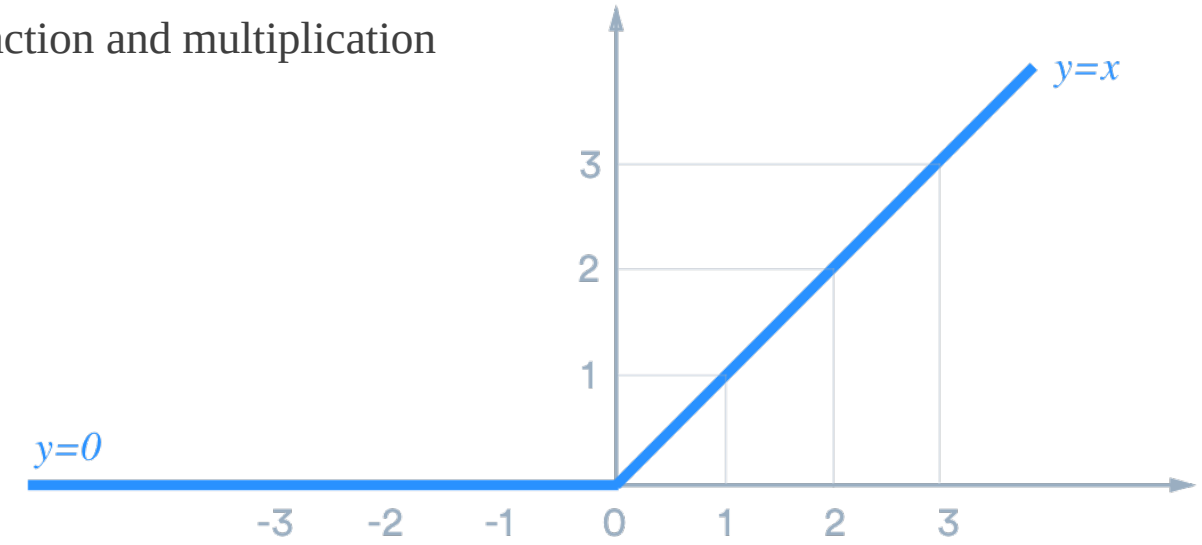
- Training
  - Requires a huge dataset of training images
  - Mathematically modify weights to fit training examples
  - Takes up a lot of time



- 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 (μ) = 3 | Error vector with small coefficients

[1] Gentry, Craig, Sahai, Amit, Waters, Brent 2013

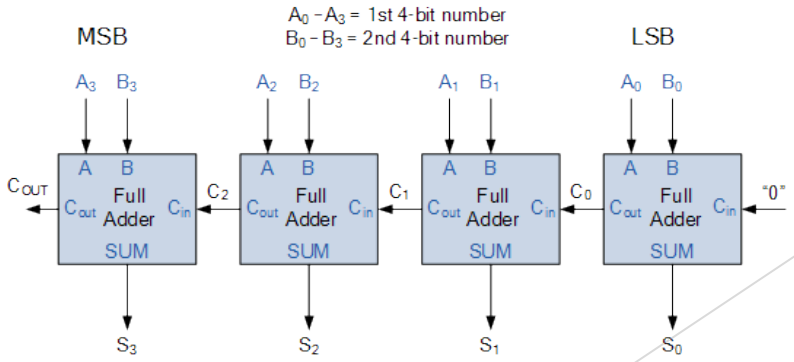
Constraints of FHE

- Gentry's Scheme makes multiplication and addition homomorphic in practice, but with the restriction that $\mu \epsilon \{0, 1\}$
  - 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)
  - NOT(A) = 1 – A
  - 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

Add/Subtract	DONE
Multiplication	DONE
Scalar Division	DONE
ReLU	DONE
Max Pool	DONE



Example Optimization - ReLU

- $\mathrm{ReLU}(x) = \frac{x+|x|}{2}$ requires the very costly operations of addition and division. However, our optimized version of $\mathrm{ReLU}(x)$ requires only one efficient operation to be performed and is even more efficient than just $|x|$
- We are using a ones-complement scheme. Let $x_1, x_2, \ldots, x_n$ be the bits of x, with x_1 being the sign bit. Let $r_1, r_2, \ldots, r_n$ be the bits of the result.
  - Slow ReLU: as described above, $\frac{x+|x|}{2}$
  - Fast ReLU: $r_i = \mathrm{OR}(x_1, x_i)$ for all $1 \leq i \leq n$

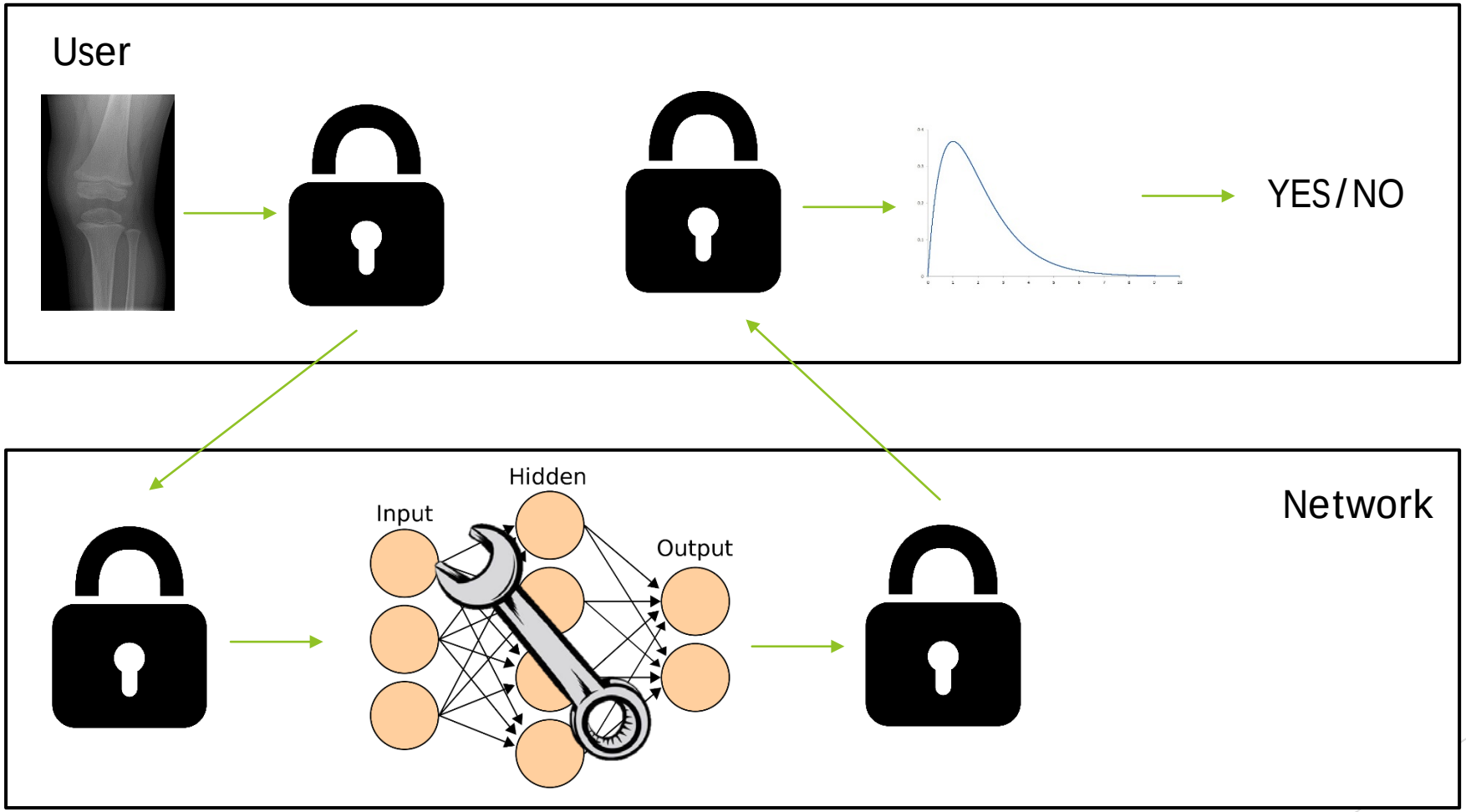
	Fast ReLU	Slow ReLU	Abs	Add	Mult
Time (s)	0.206	53.389	0.495	1.509	51.384
Time (relative to Fast ReLU)	1.0	259.2	2.4	7.3	249.4

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: $\frac{10}{1} \div \frac{5}{2} = \frac{(10)(2)}{(1)(5)} = \frac{20}{5}$
  - 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



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

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

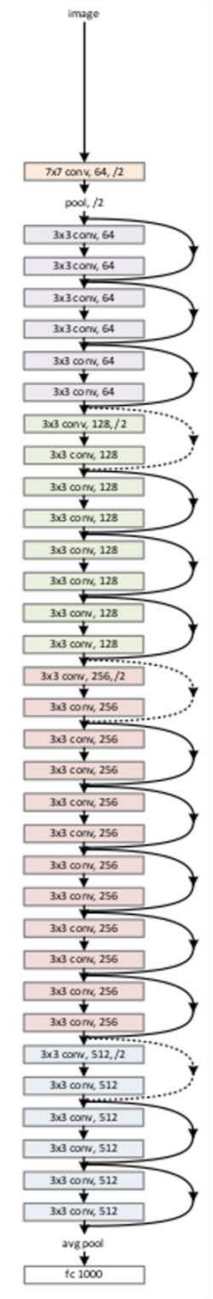
[1] Leng, Duo, Li, Zhu, Jin 2017
[2] Meng, Gu, Zhang, Wu 2017

Quantization of ResNet

Number of Bits	Top-1 Accuracy
2	72.6%
3	72.99%
4	73.64%
5	73.71%
no quantization	74.28%

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

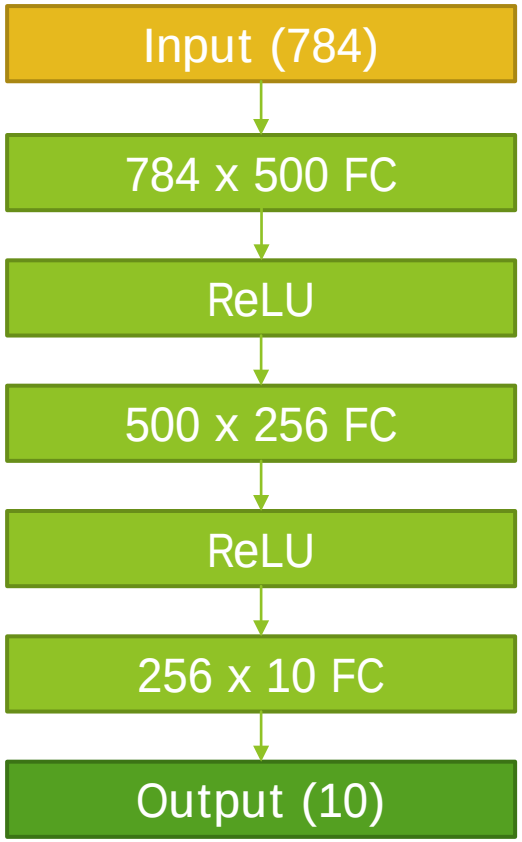
- Representation of values that allows fast computations
- Weights can be compressed without loss in accuracy
- We can avoid overflow while operating on LP values

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

Net Type	Top-1 Accuracy
MNIST Net	98.19%
8-bit MNIST Net	98.00%

- 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

Enter Gentry's Scheme! [source]

- 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.

Enter Binary Logic!

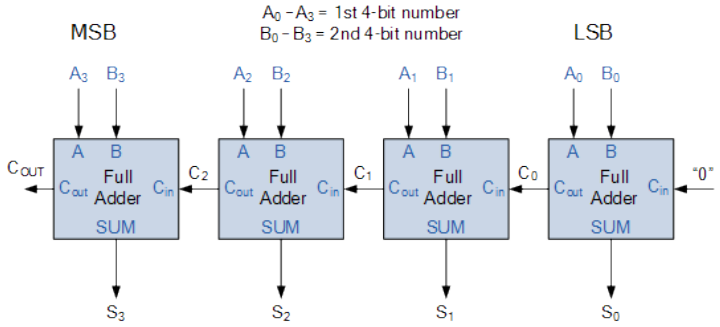
- If we have two one-bit ciphers A and B, then NAND(A, B) = 1 – AB. Since 1 is represented by the identity matrix, and addition, subtraction, and multiplication are homomorphic, NAND is also homomorphic. Equivalently, Gentry's scheme is fully homomorphic.
- Unfortunately, stringing together NAND gates always works in theory, but in practice, can be very inefficient and slow…

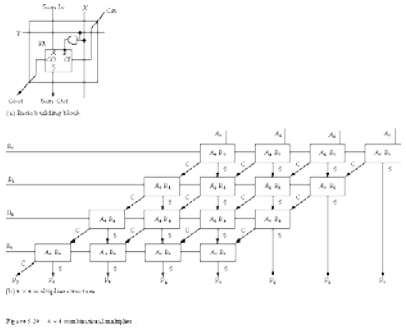
Lattice-Based Cryptography is Somewhat Homomorphic

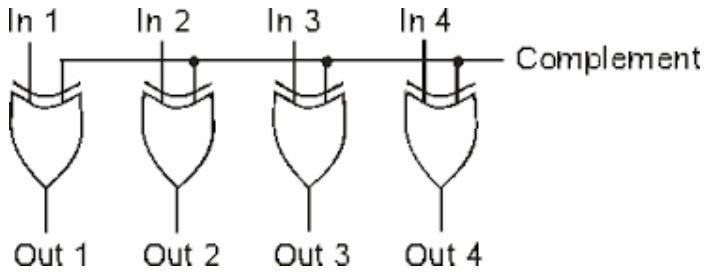
- Somewhat Homomorphic Encryption: similar to FHE but there is a small error which increases with each operation.
- Eigenvalues add and multiply corresponding to matrix additions and multiplications, very useful for homomorphic encryption!
- Since the plaintext and secret key are not exact, they are only *somewhat homomorphic* instead of *fully homomorphic*.
  - Multiplicative: $B^{(2^{\wedge}L)}$ 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.

Low Precision Weights

Given weights $W \in \mathbb{R}^m$ and N bits, we want

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

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

Let,

$$\omega_0 \coloneqq \frac{\max_{x \in W} |x|}{2^{N-1} - 1}$$

Then,

$$G_{k+1} \coloneqq \prod_{\{0, \pm 1, \pm 2, \dots, \pm(2^{N-1}-1)\}} \frac{W}{\omega_k}$$

$$\omega_{k+1} \coloneqq \frac{W^T G_{k+1}}{G_{k+1}^T G_{k+1}}$$

($\prod$ denotes the euclidean projection operator)

Matrix Addition

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

$$\gamma = 2\alpha$$

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

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

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 α and A while training,

$$M_B \coloneqq \max_{x \in A} |x|$$

$$M \coloneqq \rho M + (1 - \rho) M_B \qquad (0 < \rho < 1)$$

$$\alpha \coloneqq \frac{\alpha M_B}{2^{N-1} - 1}$$

$$A \coloneqq \left[\left(\frac{2^{N-1} - 1}{M_B} \right) A \right]$$

Inference With Scaling Layers

- LP values are encrypted

When it recieves inputs α and A during inference,

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

$$A \coloneqq \left[\left(\frac{2^{N-1}-1}{M}\right)A\right]$$

Two-Phase Cross Product

Given vectors $A, B \in \{0, \pm 1, \pm 2, \dots, \pm(2^{N-1} - 1)\}^n$ and scalars α and β:

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

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

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,

$$M_{B1} \coloneqq \max_{0 \le i < \sqrt{n}} \sum_{0 \le j < \sqrt{n}} C_{R\,1+j+i\sqrt{n}}$$

$$M_1 \coloneqq \rho M_1 + (1-\rho) M_{B1} \qquad (0 < \rho < 1)$$

Training:

$$\gamma_A \coloneqq \frac{\gamma_R M_{B1}}{2^{N-1}-1}$$

$$C_A \coloneqq \left[\left(\frac{2^{N-1}-1}{M_{B1}} \right) C_R \right]$$

Inference:

$$\gamma_A \coloneqq \frac{\gamma_R M_1}{2^{N-1}-1}$$

$$C_A \coloneqq \left[\left(\frac{2^{N-1}-1}{M_1} \right) C_R \right]$$

Phase One Continued

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

$$C_{B\,i} = \sum_{0 \le j < \sqrt{n}} C_{A\,1+j+i\sqrt{n}}$$

$$\gamma_B \coloneqq \gamma_A$$

Phase Two

During training,

$$M_{B2} \coloneqq \sum C_B$$

$$M_2 \coloneqq \rho M_2 + (1-\rho) M_{B2} \qquad (0 < \rho < 1)$$

Training:

$$\gamma \coloneqq \frac{\gamma_B M_{B2}}{2^{N-1}-1}$$

$$C \coloneqq \sum \left[\left(\frac{2^{N-1}-1}{M_{B2}} \right) C_B \right]$$

Inference:

$$\gamma \coloneqq \frac{\gamma_B M_2}{2^{N-1}-1}$$

$$C \coloneqq \left[\left(\frac{2^{N-1}-1}{M_2} \right) C_B \right]$$