SecureNNplus: Privacy-Preserving Neural Network Training using Secure Computation

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

Leave Authors Anonymous
for Submission
City, Country
e-mail address

ABSTRACT

Neural networks are used in areas ranging from image classification to machine translation, and there are often multiple parties that contribute (possibly private/sensitive) data to the training process. Thus, there arises a need for *secure* neural networks, which can help parties evaluate a joint output (trained network) without revealing their input data.

CCS Concepts

•Computing methodologies → Computing setup; •Security and privacy → Data anonymization and sanitization; Pseudonymity, anonymity and untraceability; Privacy-preserving protocols;

INTRODUCTION

Neural networks (NN) are used in areas ranging from image classification to machine translation, and there are often multiple parties that contribute (possibly private/sensitive) data to the training process. Thus, there arises a need for *secure* NN, which can help parties evaluate a joint output (trained network) without revealing their input data.

In 2018, Wagh et al. [1] described a scenario where M parties want to jointly compute an output using N servers. These parties secret-shareX their data to these servers, which interactively and securely train the NN model. No party or server learns anything about anybody else's data.

The paper focuses on the setting $N = 3$, providing efficient protocols for a variety of NN operations: ReLU activation, ReLU's derivative, Division, Maxpool and its derivative. A number of supporting protocols are also described, including MatrixMultiplication, ShareConvert[1], etc. It is critical to note that while the protocols involve 3 servers, only 2 of them hold secret shares – the third party helps with the secure computation. All computations happen in an odd ring; secret shares that are in an even ring are converted to an odd ring.

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04…$15.00

DOI: https://doi.org/10.1145/3313831.XXXXXXX

Contributions

We describe novel secure protocols for *Sigmoid*, *softmax*, and *tanh* – popular activation functionsX that are significantly more complex than ReLU (described by [1]). The inclusion of these protocols in a secure and private setting vastly increases the practicality of the framework and enables people to convert their protocols into secure ones without having to redesign the actual internal structure of their NNs.

For this purpose, we first propose a secure protocol for exponentiation. Further, we propose protocols for computing sigmoid and softmax functions, along with their derivatives. We also provide a protocol for approximating the sigmoid function using the hard sigmoid function.

BACKGROUND

Softmax is usually applied after the last layer of a NN, inputting a vector of size k, normalizing it into a probability distribution and outputting a vector of probabilities. These probabilities are proportional to the exponential values of the input components. Given input $x_1, \cdots, x_k$, the output is: $\sigma(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_i}}$

Sigmoid takes as input a numeric value from any range and outputs a value between 0 and 1. (Another commonly used range is $[-1, 1]$, but for the purposes of our protocols, we use $[0, 1]$). The function can be interpreted mathematically as: $S(x) = \frac{1}{1+e^{-x}}$

Hard Sigmoid is a piece-wise linear approximation of the sigmoid function. It retains the original shape of sigmoid, but approximates it using three simpler functions $y = 1$, $y = 0$ and $y = 0.2.x + 0.5$.

[1] transforming a share that is in an even ring to an odd ring share.

We use the following primitives from [1] in our work:

1. Matrix Multiplication $\prod_{MatMul}(\{P_0,P_1\}P_2)$:

   Two parties, P_0 and P_1 are required to hold shares of $X \in Z_L^{m\times n}$ and $Y \in Z_L^{n\times v}$. After running this interactive secure protocol, P_0 gets $\langle X.Y\rangle_0^L$ and P_1 gets $\langle X.Y\rangle_1^L$

2. Division: Two parties, P_0 and P_1 are required to hold shares of values x_j and y_j respectively, for $jin\{0,1\}$. After running the protocol, they obtain $\langle x/y\rangle_0^L$ and $\langle x/y\rangle_1^L$ respectively.

SECURENNPLUS: PROTOCOLS

Algorithm 1 Exponentiation $\prod_{\text{Exp}}(\{P_0,P_1\},P_2)$

Input: P_0 and P_1 hold $\langle x\rangle_0^L$ and $\langle x\rangle_1^L$ (shares of a value x).
Output: P_0 and P_1 obtain $\langle c\rangle_j^L = \langle e^x\rangle_j^L$.

1: For $j \in \{0,1\}$ party P_j computes $z_j = e^{\langle x\rangle_j^L}$.
2: P_0 picks random $\langle a\rangle_0^L$ and sends $\langle a\rangle_1^L = z_0 - \langle a\rangle_0^L$ to P_1 and P_1 picks random $\langle b\rangle_1^L$ and sends $\langle b\rangle_0^L = z_1 - \langle b\rangle_1^L$ to P_0.
3: P_0,P_1,P_2 invoke $\prod_{MatMul}(\{P_0,P_1\}P_2)$ with $P_j, j \in \{0,1\}$ having inputs $(\langle a\rangle_j^L, \langle b\rangle_j^L)$ and P_j learns $\langle p\rangle_j^L = \langle a.b\rangle_j^L$.
4: P_j for $j \in \{0,1\}$ output $\langle c\rangle_j^L = \langle p\rangle_j^L + u_j$.

Algorithm 2 Softmax, $\prod_{\text{Softmax}}(\{P_0,P_1\},P_2)$

Input: P_0,P_1 hold $\{\langle z_i\rangle_0^L\}_{i\in[k]}$ and $\{\langle z_i\rangle_1^L\}_{i\in[k]}$ respectively.
Output: P_0,P_1 get $\{\langle\sigma(z_i)\rangle_0^L\}_{i\in[k]}$ and $\{\langle\sigma(z_i)\rangle_1^L\}_{i\in[k]}$ respectively where $\sigma(z_i) = \frac{e^{z_i}}{\sum_{i=1}^k e^{z_i}}$.
Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: **for** $i = 1,2,\ldots,k$ **do**
2: Parties P_0, P_1 and P_2 invoke $F_{Exp}(\{P_0,P_1\},P_2)$ with inputs $\{\langle z_i\rangle_0^L\}_{i\in[k]}$ and $\{\langle z_i\rangle_1^L\}_{i\in[k]}$ and P_0, P_1 obtain shares $\langle c_i\rangle_0^L$, $\langle c_i\rangle_1^L$ resp. of $c_i^L = e^{z_i}$.
3: **end for**
4: for $j \in \{0,1\}$, P_j calculates $S_j = \sum_{i=1}^k \langle c_i\rangle_j^L$.
5: **for** $i = 1,2,\ldots,k$ **do**
6: Parties P_0, P_1 and P_2 invoke $\prod_{Division}(\{P_0,P_1\},P_2)$ with inputs $\langle c_i\rangle_j^L$ and $\langle S\rangle_j^L$.
7: Parties P_j for $j \in \{0,1\}$ output $\{\langle\sigma(z_i)\rangle_j^L\}_{i\in[k]} + u_j$.
8: **end for**

Algorithm 3 Sigmoid, $\prod_{\text{Sigmoid}}(\{P_0,P_1\},P_2)$

Input: P_0,P_1 hold $\{\langle x\rangle_0^L\}$ and $\{\langle x\rangle_1^L\}$ respectively.
Output: P_0,P_1 get $\{\langle S(x)\rangle_0^L\}$ and $\{\langle S(x)\rangle_1^L\}$ respectively where $S(x) = \frac{e^x}{1+e^x}$.
Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: Parties P_0, P_1 and P_2 invoke $F_{Exp}(\{P_0,P_1\},P_2)$ with inputs $\langle x\rangle_0^L$ and $\langle x\rangle_1^L$ and obtain $\langle a\rangle_j^L = \langle e^x\rangle_j^L$.
2: Now, parties P_0 and P_1 compute $\langle b\rangle_j^L = \langle a\rangle_j^L + j$.
3: Now they invoke $\prod_{Division}(\{P_0,P_1\},P_2)$ with inputs $\langle a\rangle_j^L$ and $\langle b\rangle_j^L$ to obtain $\langle c\rangle_j^L$ for $j \in \{0,1\}$.
4: P_j for $j \in \{0,1\}$ output $\langle c\rangle_j^L + u_j$.

Algorithm 4 Derivative of Hard Sigmoid $\langle S'_H(x)\rangle^L$

Input: P_0, P_1 have inputs $\langle a\rangle_j^L$ for P_j for $j \in \{0,1\}$.
Output: P_0 and P_1 get $\langle S'_H(a)\rangle_0^L$ and $\langle S'_H(a)\rangle_1^L$.
Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: P_0, P_1, and P_2 run $\prod_{PC}(\{P_0,P_1\},P_2)$ with P_j for $j \in \{0,1\}$ having inputs $\langle a\rangle_j^L$ and common numbers $r_1 = 2.5$ and $r_2 = -2.5$. P_2 learns two bits b_1 and b_2 corresponding to r_1 and r_2.
2: P_2 creates shares $\langle b_1\rangle_0^L$ and $\langle b_1\rangle_1^L$. P_2 also creates shares of $\langle XOR(b_1,b_2)\rangle_0^L$ and $\langle XOR(b_1,b_2)\rangle_1^L$ and sends them to P_0 and P_1.
3: for $j \in \{0,1\}$, P_0 and P_1 independently compute:

$$0.2 \times XOR(b_1,b_2)\rangle_j^L + u_j$$

Algorithm 5 Hard Sigmoid $\langle S_H(x)\rangle^L$

Input: P_0, P_1 have inputs $\langle a\rangle_j^L$ for P_j such that $j \in \{0,1\}$.
Output: P_0, P_1 get $\langle S_H(a)\rangle_0^L$ and $\langle S_H(a)\rangle_1^L$.
Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: P_0, P_1, and P_2 run $S'_H(\{P_0,P_1\},P_2)$ with inputs $\langle a\rangle_0^L$ and $\langle a\rangle_1^L$ and obtain shares $\langle c\rangle_0^L$ and $\langle c\rangle_1^L$.
2: P_0 and P_1 run $\prod_{\text{MatMul}}(\{P_0,P_1\},P_2)$ with P_j for $j \in \{0,1\}$ having inputs $\langle a\rangle_j^L$ and $\langle c\rangle_j^L$ to learn $\langle p\rangle_j^L = \langle a\rangle_j^L \times \langle c\rangle_j^L$.
3: for $j \in \{0,1\}$ using $\langle b_1\rangle_0^L$ and $\langle b_1\rangle_1^L$ and $\langle XOR(b_1,b_2)\rangle_0^L$ and $\langle XOR(b_1,b_2)\rangle_1^L$ the obtained from running $S'_H(\{P_0,P_1\},P_2)$, P_0 and P_1 independently compute:

$$\langle p\rangle_j^L + 0.5 + \langle b_1\rangle_j^L + u_j$$

FUTURE WORK

REFERENCES

[1] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2018. SecureNN: Efficient and Private Neural Network Training. *IACR Cryptology ePrint Archive* 2018 (2018), 442.