S++: A Fast and Deployable Secure-Computation Framework for Privacy-Preserving Neural Network Training

Prashanthi Ramachandran [1] Shivam Agarwal [1] Arup Mondal [1]
Aastha Shah [1] Debayan Gupta [1]
[1] Ashoka University
prashanthi.r@alumni.ashoka.edu.in, shivam.agarwal@alumni.ashoka.edu.in, arup.mondal_phd19@ashoka.edu.in,
aastha.shah@alumni.ashoka.edu.in, debayan.gupta@ashoka.edu.in

Abstract

We introduce S++, a simple, robust, and deployable framework for training a neural network (NN) using private data from multiple sources, using secret-shared secure function evaluation. In short, consider a virtual third party to whom every data-holder sends their inputs, and which computes the neural network: in our case, this virtual third party is actually a set of servers which individually learn nothing, even with a malicious (but non-colluding) adversary.

Previous work in this area has been limited to just one specific activation function: ReLU, rendering the approach impractical for many use-cases. For the first time, we provide fast and verifiable protocols for **all** common activation functions and optimize them for running in a secret-shared manner. The ability to quickly, verifiably, and robustly compute exponentiation, softmax, sigmoid, etc., allows us to use previously written NNs without modification, vastly reducing developer effort and complexity of code. In recent times, ReLU has been found to converge much faster and be more computationally efficient as compared to non-linear functions like sigmoid or tanh. However, we argue that it would be remiss not to extend the mechanism to non-linear functions such as the logistic sigmoid, tanh, and softmax that are fundamental due to their ability to express outputs as probabilities and their universal approximation property. Their contribution in RNNs and a few recent advancements also makes them more relevant.

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 data to the training process. However, training data interaction and the resulting model may still leak a significant amount of private data. Thus, there arises a need for *secure* neural networks, which can help parties collaboratively train a neural network without revealing their private input data.

Several approaches have been proposed to overcome possible data leakage, based on secure multi-party computation (MPC). MPC is a powerful way to protect sensitive training data which uses a range of cryptographic primitives to allow multiple parties to compute a function without revealing the inputs of any individual party (beyond what is implied by the output of the function itself). However, previously proposed MPC-based schemes (Ohrimenko et al. 2016; Hunt et al. 2018; Mohassel and Zhang 2017; Wagh, Gupta, and Chandran 2018; Patra and Suresh 2020) do not support models that use exponentiation-based activation functions such as *logistic sigmoid*, *softmax*, and *tanh*. These protocols are largely restricted to ReLU and its variants, which might restrict applicability in some specific cases (as described in the motivation section).

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper, we propose *S++*, a three-party secure computation framework for secure exponentiation, and exponentiation-based activation functions such as logistic sigmoid, tanh, and softmax. This enables us to construct three-party secure protocols for training and inference of several NN architectures such that no single party learns any information about the data. *S++* is an efficient MPC-based privacy-preserving neural network training framework based on (Wagh, Gupta, and Chandran 2018) for 3PC with the activation functions *logistic sigmoid*, *tanh*, their derivatives, and *softmax*. In our setting, there are D (where D can be arbitrary) data owners who wish to jointly train a model over their data with the help of 3-servers. In the setup phase, these data owners create ”additive secret-shares” of their input data and send one share each to 2-servers. In the computation phase, the 3-servers (the 2 primary servers and the helper server) collectively run an interactive protocol to train a neural network on the data owners' data without learning any information beyond the trained model.

Contributions

We summarize our key contributions as follows:

- We first propose a secure protocol for exponentiation in the three-party setting. This protocol is based on SCALE MAMBA's (Aly et al. 2020) protocol for base 2 exponentiation. It also uses primitives from (Catrina and Saxena 2010). We modify these ideas for our three-party setting and additively secret-shared data.
- We describe novel secure protocols for the *logistic sigmoid*, *softmax*, and *tanh*—popular activation functions that are significantly more complex than ReLU (described by (Wagh, Gupta, and Chandran 2018))—along with their derivatives with the help of the above-mentioned exponentiation protocol. 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.

Organization of the paper

The remainder of this paper is organized as follows: we first go over our *motivation* for extending secure protocols to functions such as the logistic sigmoid, tanh and softmax. Then, we look at *recent work* in this area, largely in the realm of MPC. After that, we describe the *notations* used in our protocols and explain some *cryptographic primitives*. In the *Protocols* section, we delve into our *architecture* and the *supporting protocols*, specifically the supporting 3-server protocols that serve as building blocks for our main protocols. We describe our *main protocols*: exponentiation, logistic sigmoid, tanh, their derivatives, and softmax. After that, we describe the theoretical efficiency of our protocols and provide an *evaluation* of our experiments. Lastly, we describe the future plans and directions for this work.

Motivation

Logistic sigmoid and variants The logistic sigmoid is a common activation function to introduce non-linearity, where $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$. It essentially squashes a real value in [0, 1] and has been used widely over the years for its simplicity, especially in binary classification tasks with a maximum likelihood loss function. It also has important variants like the tanh and softmax:

- **Tanh as a rescaled logistic sigmoid:** The tanh is a rescaling and stretching of the logistic sigmoid that maps to [-1, 1] i.e., $\tanh(x) = 2\times\text{sigmoid}(2x) - 1$. Extending the protocol to the tanh facilitates applications in recurrent neural networks where it is widely used.
- **Logistic sigmoid as a special case of the softmax:** The softmax is often used for classification tasks, where $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_i}}$. The logistic sigmoid happens to be a special case of this function, often used for binary classification. Enabling softmax thus also extends the framework to multi-class classification problems.

Why is ReLU not enough? Despite ReLU being more computationally efficient than logistic sigmoid or tanh (Krizhevsky, Sutskever, and Hinton 2017), we argue that it is worth extending the secure computation protocol to popular functions *particularly* involving exponentiation such as logistic sigmoid, tanh, and softmax for the following reasons (with more details in the appendix):

- For classification tasks, output layers usually use softmax or logistic sigmoid because they make for more interpretable values in [0, 1] as opposed to ReLU and variations that are unbounded. In fact, though previously missing in the literature, (Asadi and Jiang 2020) provide theoretical proofs on the universal approximation of softmax used in the output layer for multiclass classification tasks as well.
- ReLU's unbounded nature limits its applicability in RNNs. Capturing long-term dependencies becomes harder with an unbounded function due to exploding gradients, as explored in (Pascanu, Mikolov, and Bengio 2013). Because of the temporal correlations captured by RNNs (by forming connections between hidden layers), long-term components add up and the norm of the gradient of the hidden layers *explode* and ReLU's unbounded nature exacerbates this problem. There have been a few notable improvements to assuage this, but even in LSTMs, the tanh often gives more consistent results (Kent and Salem 2019) and ReLU tends to require greater fine-tuning.
- 'Leaky' ReLU has been used to overcome ReLU's vanishing gradient problem. (Xu, Huang, and Li 2016) have also revised logistic sigmoid to a scaled version of it and tanh to a 'leaky' tanh. The 'leaky' version penalizes the negative part of the domain. They found it to be much faster than the tanh, it gave better results than ReLU, and is almost identical to leaky ReLU. Such potential improvements highlight the need for more secure protocols for such functions.

Though ReLU has become widely adapted in the past few years, there are still some tasks wherein sigmoids are preferred over ReLU, and it is worthwhile to extend such secure protocols for these as well.

Recent Work

There have been many privacy-preserving machine learning (PPML) frameworks for various situations, such as Decision Trees (Agrawal and Srikant 2000; Lindell and Pinkas 2000), Linear Regression (Du and Atallah 2001; Sanil et al. 2004), k-means clustering (Jagannathan and Wright 2005; Bunn and Ostrovsky 2007), SVM classification (Yu, Vaidya, and Jiang 2006; Vaidya, Yu, and Jiang 2008), and logistic regression (Slavkovic, Nardi, and Tibbits 2007). However, these cannot generalize to a number of (complex and high accuracy) standard machine learning applications.

To overcome this, SecureML (Mohassel and Zhang 2017) provided secure protocols for linear regression, logistic regression, and neural networks with linear activations, using a combination of arithmetic and garbled circuits for a single semi-honest adversary in both the 2 and 3-server models. It also provided secure protocols for logistic sigmoid and softmax in the 2-server setting (although very briefly) and a method for truncating decimal numbers. MiniONN (Liu et al. 2017) optimized the protocols of SecureML by reducing the offline cost of matrix multiplications and increasing the online cost for the 2-server model in the semi-honest adversary setting. Chameleon (Riazi et al. 2018) and Gazelle (Juvekar, Vaikuntanathan, and Chandrakasan 2018) provided secure inference protocols which are computationally secure against one semi-honest adversary in the 3-server and 2-server models. Chameleon removes expensive oblivious transfer protocols by using the third party as a dealer, while Gazelle focuses on making the linear layers more communication efficient by providing specialized packing schemes for additively homomorphic encryption schemes.

SecureNN (Wagh, Gupta, and Chandran 2018) provides secure neural network training protocols for non-linear activation functions in the 3-party setting with up to one malicious corruption and shows that the honest-majority can improve the performance by several orders of magnitude. For linear regression, logistic regression and neural networks, the problem is even more challenging as the training procedure computes many instances of non-linear activation functions such as logistic sigmoid, tanh, and softmax that are expensive to compute inside a 2 and 3-server.

SecureNN provides efficient protocols for computing the ReLU activation, maxpool and their derivatives. It suggests that the general idea can extended to other variants and piecewise linear approximations. However, (Mohassel and Zhang 2017) show experimentally that low-degree polynomial approximations, like piecewise linear approximations, are ineffective in terms of accuracy. It can be proven that hard sigmoid, a piecewise linear approximation of the logistic sigmoid, performs worse than logistic sigmoid, especially in the case of linear regression (Maksutov 2018). SecureML also provides higher-order approximations of logistic sigmoid and softmax using a combination of ReLU functions in place of exponentiation. However, they do not provide a detailed algorithm and find the running time in the offline phase to be high.

To overcome this research gap, we propose *S++*, based on (Wagh, Gupta, and Chandran 2018), for the 3-server setting with efficient protocols for *logistic sigmoid* and *tanh* and their derivatives, as well as the *softmax*. This can be extended to the derivative of the softmax as well.

Cryptographic Primitives

Notation

In our work, we use 2-out-of-2 additive sharing over the even ring Z_L where $L = 2^l$. For our purposes, we use the same ring as (Wagh, Gupta, and Chandran 2018), i.e., $Z_{2^{64}}$. The 2-out-of-2 additive shares are represented by $\langle x \rangle_0^L$ and $\langle x \rangle_1^L$. where L represents the ring Z_L over which the value x has been shared.

We use the notation $\langle a \rangle_j$ to denote Party P_j's additive secret share of the value a, and $\lfloor x \rfloor$ to denote the floor of the value x. The notation $(p_0, p_1, \cdots, p_{n-1})$ is used to denote the bits of a n-bit value, p.

Further, we use the following primitives from (Wagh, Gupta, and Chandran 2018) in our work:

1. Matrix Multiplication of two secret shared matrices represented by $\mathcal{F}_{MatMul}(\{P_0, P_1\}P_2)$.
2. Division of a two secret shared values represented by $\mathcal{F}_{Division}(\{P_0, P_1\}P_2)$.

These protocols are described in the following section.

Protocols of S++

Architecture

S++ works with the 3-server setting similar to SecureNN (Wagh, Gupta, and Chandran 2018). In this setting, there are two primary servers and one helper server. The two primary servers hold additive secret shares of the data. Let P_0, P_1, P_2 be the three servers and C be a set of n participants $\{C_1, C_2, \ldots C_n\}$, where the ith party C_i holds its own private dataset, $\mathbb{D}_i$. $\mathcal{M}_{nn}$ is the neural network model to be trained on the parties' private data. The participants $\{C_1, C_2, \ldots C_n\}$ split their data into 2-out-of-2 additive secret shares and give one share to each of the two servers P_0 and P_1. The third server, P_2, similar to (Wagh, Gupta, and Chandran 2018), is crucial to the protocols, but does not hold any data. We use the same integer representation for fixed point numbers as (Wagh, Gupta, and Chandran 2018). However, to tackle any overflow that can render the exponentiation protocol impractical for use, we also consider extending the integer ring to Z_{128}. S++ provides security against one semi-honest adversary and up to one malicious corruption.

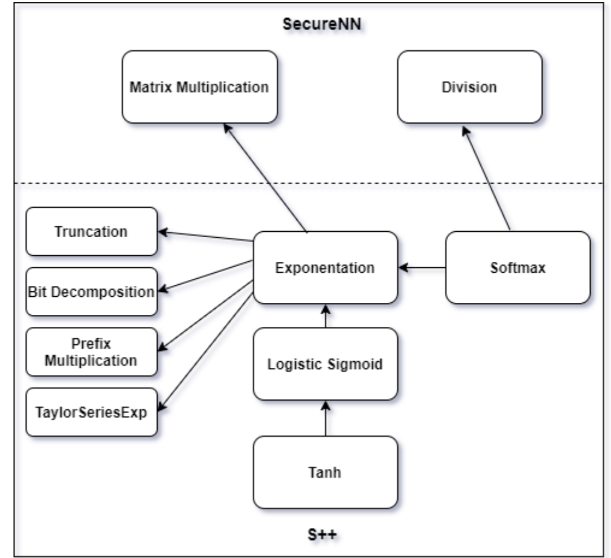

Figure 1: Function Dependencies in S++.

Supporting Protocols

Matrix Multiplication ($\mathcal{F}_{MatMul}(\{P_0, P_1\}P_2)$): In S++, this function from (Wagh, Gupta, and Chandran 2018) has been used for multiplying fixed point values (represented as unsigned integers in the Z_L ring) by considering the values as 1×1 matrices.

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 $\langle X.Y \rangle_1^L$.

Division ($\mathcal{F}_{Division}(\{P_0, P_1\}P_2)$): This function, also from (Wagh, Gupta, and Chandran 2018), is used to perform secure division on secret shared values. In this protocol, two parties, P_0 and P_1 are required to hold shares of values x_j and y_j respectively, for $j in \{0, 1\}$. After running the protocol, they obtain $\langle x/y \rangle_0^L$ and $\langle x/y \rangle_1^L$ respectively.

Taylor Series Expansion ($\mathcal{F}_{taylorExp}(\{P_0, P_1\}P_2)$; see algorithm 1): In this protocol, we compute e^x, where x is a secret fractional value. We do so by computing the value of

the Taylor expansion up to four terms. In the end of the protocol, parties P_0 and P_1 obtain shares of e^x.

Algorithm 1 Taylor Expansion, $\mathcal{F}_{taylorExp}(\{P_0, P_1\}, P_2)$

Input: P_0, P_1 hold $\{\langle x\rangle_0^L\}$ and $\{\langle x\rangle_1^L\}$ respectively where $x < 1$.

Output: P_0 and P_1 obtain $\langle y\rangle_j^L$ = $\langle e^x\rangle_j^L$.

Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: Parties P_0 and P_1 compute $\langle c\rangle_j^L = j$.
2: Parties P_0 and P_1 compute $\langle numerator\rangle_j^L = \langle x\rangle_j^L$ and $denominator = 1$.
3: Now, Parties P_0 and P_1 compute $\langle c\rangle_j^L = \langle c\rangle_j^L + \langle numerator\rangle_j^L$.
4: **for** $i = 2, \ldots, 4$ **do**
5: Parties P_0, P_1 and P_2 invoke $\mathcal{F}_{MatMul}(\{P_0, P_1\}P_2)$ with inputs $\langle numerator\rangle_j^L$ and $\langle x\rangle_j^L$ to obtain $\langle numerator\rangle_j^L$ for $j \in \{0, 1\}$.
6: $denominator = denomiator \times i$
7: Parties P_0 and P_1 compute $\langle c\rangle_j^L = \langle c\rangle_j^L + \frac{\langle numerator\rangle_j^L}{denominator}$
8: **end for**
9: P_j for $j \in \{0, 1\}$ output $\langle c\rangle_j^L$ + u_j.

Exponentiation ($\mathcal{F}_{Exp}(\{P_0, P_1\}, P_2)$); see algorithm 2: In this protocol, we compute e^x, where x is secret. This protocol is based on SCALE MAMBA's (Aly et al. 2020) base-2 exponentiation protocol. It uses primitives like $\mathcal{F}_{Trunc}$, $\mathcal{F}_{BitDecomp}$, $\mathcal{F}_{PreMult}$ from (Catrina and Saxena 2010).

While (Aly et al. 2020) uses the polynomial $P_{1045}(X)$ from (Hart 1978) to compute the exponentiation for the fractional part of a number, we use a secure protocol to compute the output using the Taylor series expansion of e^x. We introduce function $\mathcal{F}_{taylorExp}$ defined above for this purpose.

This protocol first uses $\mathcal{F}_{Trunc}$ to split the input a into its integer (a_{int}) and fractional (a_{frac}) parts. After that, it evaluates $e^{a_{\text{int}}}$ using primitives $\mathcal{F}_{BitDecomp}$ and $\mathcal{F}_{PreMult}$ described above. It further evaluates $e^{a_{\text{frac}}}$ using $\mathcal{F}_{taylorExp}$. It then multiplies these two values to obtain $e^{a_{\text{int}}+a_{\text{frac}}} = e^a$.

Algorithm 2 Exponentiation $\mathcal{F}_{Exp}(\{P_0, P_1\}, P_2)$

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 $\langle y\rangle_j^L$ = $\langle e^x\rangle_j^L$.

1: For $j \in \{0, 1\}$, party P_j calls $\mathcal{F}_{Trunc}(\{P_0, P_1\})$ to obtain $\langle a\rangle_j^L$, which is the jth share of $\lfloor x \rfloor$.
2: For $j \in \{0, 1\}$, party P_j gets the fractional part of x by locally computing:
$$\langle b\rangle_j^L = \langle x\rangle_j^L - \langle a\rangle_j^L$$
3: For $j \in \{0, 1\}$ party P_j invokes $\mathcal{F}_{BitDecomp}(\{P_0, P_1\})$ to obtain $(\langle c_0\rangle_j^L, \langle c_1\rangle_j^L, \cdots, \langle c_{m-1}\rangle_j^L)$: shares of $(x_0, x_1, \cdots, x_{m-1})$, where x is a m-bit number.
4: **for** $i = 0, 1, \ldots, m-1$ **do**
5: P_j for $j \in \{0, 1\}$ computes $\langle v_i\rangle_j^L = e^{2^i}.(\langle c_i\rangle_j^L) + j - (\langle c_i\rangle_j^L)$.
6: **end for**
7: P_j for $j \in \{0, 1\}$ invokes $\mathcal{F}_{PreMult}(\{P_0, P_1\})$ to get $\langle m\rangle_j^L$.
8: P_j for $j \in \{0, 1\}$ invokes $\mathcal{F}_{Taylor}(\{P_0, P_1\})$ to get $\langle n\rangle_j^L$
9: Finally, P_j for $j \in \{0, 1\}$ invokes $\mathcal{F}_{MatMul}(\{P_0, P_1\}P_2)$ with inputs $\langle m\rangle_j^L$ and $\langle n\rangle_j^L$ to obtain share $\langle m\rangle_j^L$ of:
$$y = m \times n.$$

Main Protocols

Our main protocols include logistic sigmoid ($\mathcal{F}_{Sigmoid}$; described in algorithm 3), tanh ($\mathcal{F}_{Tanh}$; described in algorithm 5), and their derivatives; see algorithms 4 and 6, as well as the softmax function ($\mathcal{F}_{Softmax}$; described in algorithm 7).

Sigmoid, $\mathcal{F}_{Sigmoid}(\{P_0, P_1\}, P_2)$; see algorithm 3): This protocol can be used to securely compute logistic sigmoid. Parties P_0, P_1 and P_2 invoke $\mathcal{F}_{Exp}$ with shares of the input x to obtain output shares of e^x. They then jointly compute shares of $1+e^x$. They use (Wagh, Gupta, and Chandran 2018)'s secure division protocol to obtain $\sigma(x) = \frac{e^x}{1+e^x}$.

Algorithm 3 Sigmoid, $\mathcal{F}_{Sigmoid}(\{P_0, P_1\}, P_2)$

Input: P_0, P_1 hold $\{\langle x\rangle_0^L\}$ and $\{\langle x\rangle_1^L\}$ respectively.

Output: P_0, P_1 get $\{\langle \sigma(x)\rangle_0^L\}$ and $\{\langle \sigma(x)\rangle_1^L\}$ respectively where $\sigma(x) = \frac{e^x}{1+e^x}$.

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: P_0, P_1 and P_2 invoke $\mathcal{F}_{Division}(\{P_0, P_1\}, P_2)$ 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.

Derivative of Sigmoid, $\mathcal{F}^{\mathcal{D}}{}_{Sigmoid}(\{P_0, P_1\}, P_2)$; see algorithm 4): This protocol can be used to securely compute the derivative of the logistic sigmoid. We use the idea that $\frac{dS(x)}{d(x)} = \sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$. In the end of the protocol, parties P_0 and P_1 obtain the shares of $\sigma'(x)$.

Algorithm 4 Derivative of Sigmoid $\langle \mathcal{F^D}_{sigmoid}(x) \rangle^L$

Input: P_0, P_1 have inputs $\langle x \rangle_j^L$ for P_j such that $j \in \{0,1\}$.

Output: P_0, P_1 get $\langle \sigma'(c) \rangle_0^L$ and $\langle \sigma'(c) \rangle_1^L$ where $\sigma'(c) = \sigma(x)(1-\sigma(x))$

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 invoke $\mathcal{F}_{Sigmoid}(\{P_0, P_1\}, P_2)$ with P_j for $j \in \{0,1\}$ having input $\langle x \rangle_j^L$ to learn $\langle a \rangle_j^L = \langle \sigma(x) \rangle_j^L$.
2: P_j computes $\langle a \rangle_j^L = -1 \times \langle x \rangle_j^L + j$ for $j \in \{0,1\}$
3: P_j computes $\langle b \rangle_j^L = \langle a \rangle_j^L$ for $j \in \{0,1\}$
4: P_0, P_1 and P_2 invoke $\mathcal{F}_{MatMul}(\{P_0, P_1\}, P_2)$ with P_j for $j \in \{0,1\}$ having inputs $\langle a \rangle_j^L$ and $\langle b \rangle_j^L$ to learn $\langle c \rangle_j^L = \langle a \rangle_j^L \times \langle b \rangle_j^L$.
5: P_j for $j \in \{0,1\}$ output:

$$\langle c \rangle_j^L + \langle u \rangle_j^L$$

Tanh, $\mathcal{F}_{Tanh}(\{P_0, P_1\}, P_2)$; see algorithm 5): This protocol can be used to securely compute the tanh function. For this protocol, we use the idea that $tanh(x)$ is related to $\sigma(x)$ as $tanh(x) = 2\sigma(2x) - 1$.

Algorithm 5 Tanh $\langle \mathcal{F}_{Tanh}(x) \rangle^L$

Input: P_0, P_1 have inputs $\langle x \rangle_j^L$ for P_j such that $j \in \{0,1\}$.

Output: P_0, P_1 get $\langle tanh(c) \rangle_0^L$ and $\langle tanh(c) \rangle_1^L$ where $tanh(c) = \frac{2}{1+e^{-2x}} - 1$

Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: P_j computes $\langle a \rangle_j^L = 2 \times \langle x \rangle_j^L$ for $j \in \{0,1\}$
2: P_0, $P_1 and P_2$ invoke $\mathcal{F}_{Sigmoid}(\{P_0, P_1\}, P_2)$ with P_j for $j \in \{0,1\}$ having input $\langle a \rangle_j^L$ to learn $\langle p \rangle_j^L = \langle \sigma(a) \rangle_j^L$.
3: P_j for $j \in \{0,1\}$ output:

$$2 \times \langle P \rangle_j^L - j + \langle u \rangle_j^L$$

Derivative of Tanh, $\mathcal{F^D}_{Tanh}(\{P_0, P_1\}, P_2)$; see algorithm 6): This protocol can be used to securely compute the derivative of the tanh function. For this protocol, we use the idea that $tanh'(x)$ is related to $\sigma'(x)$ as $tanh'(x) = 4\sigma'(2x)$.

Softmax, $\mathcal{F}_{Softmax}(\{P_0, P_1\}, P_2)$; see algorithm 6): This protocol can be used to securely compute the softmax function. The two parties compute the numerator e^{z_i} and the denominator $\sum_{i=1}^{k} e^{z_i}$ and invoke the $\mathcal{F}_{Division}$ function to perform secure division.

Empirical Evaluation

Implementation and Experimental Setup

Our system is implemented in C++ using standard libraries. To be able to execute machine learning protocols, we use

Algorithm 6 Derivative of Tanh $\langle \mathcal{F^D}_{Tanh}(x) \rangle^L$

Input: P_0, P_1 have inputs $\langle x \rangle_j^L$ for P_j such that $j \in \{0,1\}$.

Output: P_0 and P_1 get $\langle \tanh'(c) \rangle_0^L$ and $\langle \tanh'(c) \rangle_1^L$ respectively.

Common Randomness: P_0 and P_1 hold random shares of zero - u_0 and u_1 respectively.

1: P_0 and P_1 compute $\langle a \rangle_j^L = 2 \times \langle x \rangle_j^L$.
2: P_0, P_1, P_2 invoke $\mathcal{F^D}_{sigmoid}$ with $\langle a \rangle_j^L$ to obtain shares $\langle b \rangle_j^L$.
3: P_0 and P_1 output:

$$4.\langle b \rangle_j^L + \langle u \rangle_j^L$$

Algorithm 7 Softmax, $\mathcal{F}_{Softmax}(\{P_0, P_1\}, P_2)$

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 $\{\langle s_{max}(z_i) \rangle_0^L\}_{i \in [k]}$ and $\{\langle s_{max}(z_i) \rangle_1^L\}_{i \in [k]}$ respectively where $s_{max}(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_i}}$.

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 $\mathcal{F}_{Division}(\{P_0, P_1\}, P_2)$ with inputs $\langle c_i \rangle_j^L$ and $\langle S \rangle_j^L$.
7: Parties P_j for $j \in \{0,1\}$ output $\{\langle_{max}(z_i) \rangle_j^L\}_{i \in [k]} + u_j$.
8: **end for**

fixed-point representation in the integer ring $\mathbb{Z}_{64}$. This allows us to compute precise values using our machine learning protocols. The fixed-arithmetic used in our implementation is the same as (Wagh, Gupta, and Chandran 2018) which borrowed the idea from (Mohassel and Zhang 2017).

Exponentiation is complicated and may overflow the integer ring. So, we consider the ring $\mathbb{Z}_{128}$ as proposed by (Aly et al. 2020). So far, we have implemented the Taylor series exponentiation (refer to step 7 of Algorithm-1) which works for values less than 1. We provide benchmarks for our logistic sigmoid, tanh and softmax functions, their derivatives and the Taylor series exponentiation in the secure setting with input vectors of varying sizes. These benchmarks include the execution time and the total communication (in MB) of these protocols.

We run our protocols in a LAN setting on an Intel i5 7th Gen with 8GB RAM. The average bandwidth measured was 40Mb/s for download and 9.2Mb/s for upload and the average ping was 12ms.

Experimental Results

We show the average time taken by the protocols in Table 1.

Protocol	Dimension	Time(s)	Comm.(mb)
$\mathcal{F}_{Exp}$	64x16	0.08	0.025
	128x128	2.134	0.393
	576x20	0.882	0.276
$\mathcal{F}_{Sigmoid}$	64x16	0.252	2.58
	128x128	5.631	41.288
	576x20	2.615	29.03
$\mathcal{F}_{tanh}$	64x16	0.275	2.58
	128x128	5.32	41.288
	576x20	2.613	29.03
$\mathcal{F}_{Softmax}$	64x16	0.324	2.58
	128x128	5.438	41.288
	576x20	2.617	29.03
${\mathcal{F}^{\mathcal{D}}}_{sigmoid}$	64x16	0.464	2.597
	128x128	8.033	41.55
	576x20	4.121	29.214
${\mathcal{F}^{\mathcal{D}}}_{tanh}$	64x16	0.383	2.58
	128x128	4.465	41.288
	576x20	2.84	29.03
$\mathcal{F}_{taylorExp}$	64x16	0.032	0.005
	128x128	0.092	0.079
	576x20	0.427	0.055

Table 1: Benchmarks in LAN setting on i5 7th Gen

Conclusion and Future Work

Previous work in secure computation of neural networks has been able to achieve reasonable efficiency for many real world applications. However, the lack of options in the choice of protocols that these implementations can run can limit the capacity of activities the user is able to perform in the secure setting.

Through the addition of the exponentiation function, it becomes possible to add a large number of secure activation functions that were previously impossible to compute. We have shown logistic sigmoid, tanh, softmax and their derivatives as examples of such functions. We implement these functions with similar overheads as the previously proposed activation functions.

In its current form, the exponentiation function overflows for small values making it impractical for usage in regular neural network. This also makes us unable to use the functions that call the exponentiation function, as part of their execution, in practical settings.

Exponentiation protocols in a fixed-point MPC setting have been provided by (Aly and Smart 2019) and implemented by (Aly et al. 2019). Ideas from these protocols can be used to avoid using a workaround that would involve increasing the size of shares to accommodate the overflow.

The future work on S++ will be focused on dealing with the overflow such that the exponentiation function can handle the magnitude of the values that it would encounter in a regular neural network. Once this is solved, the other functions that we have proposed, that use exponentiation, will automatically become suitable for practical usage.

References

Agrawal, R.; and Srikant, R. 2000. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 439–450.

Aly, A.; Cong, K.; Cozzo, D.; Keller, M.; Orsini, E.; Rotaru, D.; Scherer, O.; Scholl, P.; Smart, N.; Tanguy, T.; et al. 2020. SCALE–MAMBA v1. 10: Documentation .

Aly, A.; Orsini, E.; Rotaru, D.; Smart, N. P.; and Wood, T. 2019. Zaphod: Efficiently combining lsss and garbled circuits in scale. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 33–44.

Aly, A.; and Smart, N. P. 2019. Benchmarking privacy preserving scientific operations. In *International Conference on Applied Cryptography and Network Security*, 509–529. Springer.

Asadi, B.; and Jiang, H. 2020. On Approximation Capabilities of ReLU Activation and Softmax Output Layer in Neural Networks. *arXiv preprint arXiv:2002.04060* .

Bunn, P.; and Ostrovsky, R. 2007. Secure two-party k-means clustering. In *Proceedings of the 14th ACM conference on Computer and communications security*, 486–497.

Catrina, O.; and Saxena, A. 2010. Secure Computation with Fixed-Point Numbers. In *International Conference on Financial Cryptography and Data Security*. Springer.

Du, W.; and Atallah, M. J. 2001. Privacy-preserving cooperative scientific computations. In *csfw*, 0273. Citeseer.

Hart, J. F. 1978. Computer Approximations. In *Krieger Publishing Co., Inc.* Springer.

Hunt, T.; Song, C.; Shokri, R.; Shmatikov, V.; and Witchel, E. 2018. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961* .

Jagannathan, G.; and Wright, R. N. 2005. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 593–599.

Juvekar, C.; Vaikuntanathan, V.; and Chandrakasan, A. 2018. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 1651–1669.

Kent, D.; and Salem, F. 2019. Performance of three slim variants of the long short-term memory (LSTM) layer. In *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, 307–310. IEEE.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2017. Imagenet classification with deep convolutional neural networks. *Communications of the ACM* 60(6): 84–90.

Le, Q. V.; Jaitly, N.; and Hinton, G. E. 2015. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941* .

Lindell, Y.; and Pinkas, B. 2000. Privacy preserving data mining. In *Annual International Cryptology Conference*, 36–54. Springer.

Liu, J.; Juuti, M.; Lu, Y.; and Asokan, N. 2017. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 619–631.

Maksutov, R. 2018. Deep study of a not very deep neural network. Part 2: Activation functions. https://towardsdatascience.com/deep-study-of-a-not-very-deep-neural-network-part-2-activation-functions-fd9bd8d406fc. Accessed: 2020-11-06.

Mohassel, P.; and Zhang, Y. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, 19–38. IEEE.

Ohrimenko, O.; Schuster, F.; Fournet, C.; Mehta, A.; Nowozin, S.; Vaswani, K.; and Costa, M. 2016. Oblivious multi-party machine learning on trusted processors. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 619–636.

Pascanu, R.; Mikolov, T.; and Bengio, Y. 2013. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, 1310–1318.

Patra, A.; and Suresh, A. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *arXiv preprint arXiv:2005.09042* .

Riazi, M. S.; Weinert, C.; Tkachenko, O.; Songhori, E. M.; Schneider, T.; and Koushanfar, F. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 707–721.

Sanil, A. P.; Karr, A. F.; Lin, X.; and Reiter, J. P. 2004. Privacy preserving regression modelling via distributed computation. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 677–682.

Slavkovic, A. B.; Nardi, Y.; and Tibbits, M. M. 2007. " Secure" Logistic Regression of Horizontally and Vertically Partitioned Distributed Databases. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, 723–728. IEEE.

Vaidya, J.; Yu, H.; and Jiang, X. 2008. Privacy-preserving SVM classification. *Knowledge and Information Systems* 14(2): 161–178.

Wagh, S.; Gupta, D.; and Chandran, N. 2018. SecureNN: Efficient and Private Neural Network Training. *IACR Cryptology ePrint Archive* 2018: 442.

Xu, B.; Huang, R.; and Li, M. 2016. Revise saturated activation functions. *arXiv preprint arXiv:1602.05980* .

Yu, H.; Vaidya, J.; and Jiang, X. 2006. Privacy-preserving svm classification on vertically partitioned data. In *Pacific-asia conference on knowledge discovery and data mining*, 647–656. Springer.

Appendix

Universal approximation of softmax: (Asadi and Jiang 2020) provide theoretical proofs on the universal approximation of ReLU used in hidden layers of deep networks and softmax used in the output layer for multiclass classification tasks. They find that such a proof for the softmax is missing in the literature despite it being widely used for these tasks, and provide their own to prove the power of the softmax used in output layers to approximate indicator functions.

ReLU and gradients: (Pascanu, Mikolov, and Bengio 2013) delve into the exploding and vanishing gradients of activation functions that tend to cause problems in recurrent neural networks because of the temporal correlations they capture (by forming connections between hidden layers). Long-term components add up and the norm of the gradient of the hidden layers *explode*. ReLU's unbounded nature exacerbates this problem. There have been a few notable improvements to assuage this, namely gradient clipping, identity initialization (Le, Jaitly, and Hinton 2015), and gated RNNs like LSTMs and GRUs. However, even in LSTMs, the tanh often gives more consistent results ((Kent and Salem 2019)) and ReLU tends to require greater fine-tuning.

The Leaky Tanh: (Xu, Huang, and Li 2016) propose a 'leaky' tanh that penalizes the negative part i.e., instead of the regular tanh function, the penalized tanh behaves like this:

$$penalized_tanh(x) = \begin{cases} tanh(x) & x > 0 \\ a.tanh(x) & otherwise \end{cases} \quad (1)$$

where $a \in (0, 1)$. They also found this variation to be two times faster than the tanh.