On MPC-friendly Softmax

Marcel Keller and Ke Sun

CSIRO's Data61

Abstract. Softmax is widely used in deep learning to map some representation to a probability distribution. As it is based on the exponential function, which is relatively expensive in multi-party computation, Mohassel and Zhang (S&P, 2017) proposed a simpler replacement based on ReLU (the maximum of the input and zero) to be used in secure computation. Later works (e.g., Wagh et al., PETS 2019 and 2021) used the softmax replacement not for computing the output probability distribution but for approximating the gradient in backpropagation. In this work, we analyze the two uses of the replacement and compare them to softmax, both in terms of accuracy and cost in multi-party computation. We found that the replacement only provides a significant speed-up for a one-layer network, while it always reduces accuracy, sometimes significantly. Thus, we conclude that its usefulness is limited, and one should use the original softmax function instead. We also present a novel protocol for secure exponentiation that reduces communication up to fourfold while preserving accuracy when used for softmax.

Keywords: Privacy-preserving machine learning · Multiparty computation · Deep learning

1 Introduction

Multi-party computation (MPC) is a technology for collaborative computation without individual parties learning the input or intermediate data. As such, it has been proposed as a key tool for privacy-preserving machine learning. Mohassel and Zhang [MZ17] have initiated a line of work proposing concrete implementations of deep learning algorithms. A deep neural classifier applies parameterized compositions of functions (called layers) to input data to make predictions. The *loss*, which is a function of the data as well as all parameters $\boldsymbol{\Theta}$ across the layers, is used to measure the prediction error of the model. Gradients of the loss with respect to $\boldsymbol{\Theta}$ are computed using auto-differentiation (AD), which is the modern standard approach to apply the chain rule over a computational graph to obtain the loss derivatives efficiently and exactly. Subsequently, all parameters $\boldsymbol{\Theta}$ are updated based on a numerical optimization algorithm, such as variants of gradient descent, which consists of an iterative process to adjust $\boldsymbol{\Theta}$ in directions opposite to the loss gradient, to minimize the loss.

A desirable component in deep learning is a function that maps $\boldsymbol{x} \in \mathbb{R}^L$ ($L \geq 2$) to an L-entry probability vector $\boldsymbol{p}$ such that the largest entry of the input gets assigned the highest probability. This is achieved by the softmax function, defined as

$$p_i := \frac{\exp(x_i)}{\sum_{j=0}^{L-1} \exp(x_j)}. \tag{1}$$

In this paper, we denote by $\boldsymbol{x}$ the prediction of the classifier network given by the linear output of the final layer (i.e., logits). One has to keep in mind that $\boldsymbol{x}$ depends on the

E-mail: marcel.keller@data61.csiro.au,mks.keller@gmail.com (Marcel Keller), ke.sun@data61.csiro.au (Ke Sun)

input data and is usually parameterized by the neural network weights and biases $\boldsymbol{\Theta}$ so that $\boldsymbol{x} := \boldsymbol{x}(\boldsymbol{\Theta})$. It is easy to check that the output $\boldsymbol{p} = (p_0, p_1, \cdots)$ defines a probability distribution. Indeed, by Equation 1, all entries of $\boldsymbol{p}$ are strictly positive and sum up to one. The name "softmax" is derived from the fact that it can be seen as a smoothed version of the function that outputs 1 for the maximum input entry and 0 for all others. This simple maximum function is not differentiable, which makes it unsuitable for deep learning.

The underlying mathematics of MPC primarily offer modular addition and multiplication as core operations, with even comparison not directly available for most schemes. While it is possible to build non-linear computation using these, the relative cost compared to the core operations is often higher than with microprocessors. For example, Intel's AVX-512 can approximate powers of two for floating-point values in ten clock cycles [Fog25] compared to one clock cycle for basic operations such as multiplication. This is generally not the case for MPC, as shown in Table 2, where the cost of exponentation is at least 30 times the cost of multiplication. Unlike comparison and division [CS10], that have been studied for over a decade, concrete MPC algorithms for exponentiation have emerged only more recently [AS19, RRG+21].

This fact led Mohassel and Zhang [MZ17] to propose a replacement for softmax that only requires comparison and division:

$$\tilde{p}_i := \begin{cases} \frac{\mathsf{ReLU}(x_i)}{\sum_i \mathsf{ReLU}(x_i)}, & \text{if } \sum_i \mathsf{ReLU}(x_i) > 0 \\ 1/L, & \text{otherwise} \end{cases} \tag{2}$$

where ReLU (Rectified Linear Unit) [GBB11] is defined as

$$\mathsf{ReLU}(x) := \begin{cases} x, & \text{if } x > 0 \\ 0. & \text{otherwise} \end{cases}$$

Due to computational constraints, softmax alternatives proposed in the machine learning literature [MA16, PNM19] have not been widely adopted by the MPC community.

This replacement by Mohassel and Zhang was subsequently used by several works in privacy-preserving machine learning [WTB+21, WGC19, CRS20, PSSY21, KPRS22]. However, neither Mohassel and Zhang nor any of the further works explored the implications of replacing softmax. This holds for both the mathematics of the machine learning process and the effect on the accuracy. In this work, we explore these aspects, and we present an implementation of softmax in MPC that considerably improves accuracy while incurring only a moderate cost. More concretely, we show that the test error can be halved with only a 21% increase in cost.

2 Softmax in Machine Learning

We use multi-class classification as a typical example where softmax is applied in deep learning. Another broad application of softmax is the attention mechanism [VSP+17] underpinning large language models, where softmax normalizes similarity scores between items (e.g., words or tokens) in a sequence (e.g., a sentence) to determine how strongly they influence one another.

Consider recognizing hand-written digits, where the input is an image and the output is a class label from 0 to 9, signifying the digit it represents. Given the input, a deep neural network can learn a vector representation $\boldsymbol{x} = (x_0, \ldots, x_9) \in \mathbb{R}^{10}$, where a larger x_i means a higher likelihood of the input being the digit i. In order to turn $\boldsymbol{x}$ into a probability distribution (and to define a learning process), the softmax function is commonly used.

Usually, learning is implemented by minimizing a loss function. For a training sample, its loss, indicating the incorrectness of the model's prediction, is defined in terms of the

output distribution $\boldsymbol{p}$, as well as the ground truth one-hot vector $\boldsymbol{y} = (y_0, y_1, \dots)$ with $y_i = 1$ for the true class label i and $y_i = 0$ otherwise. In this paper, our expressions directly generalize to the case when $\boldsymbol{y}$ is a probability distribution [MKH19], i.e., when multiple elements of $\boldsymbol{y}$ have non-zero values. The global loss is a sum of the per-sample losses. A commonly used loss function is the cross-entropy:

$$\ell := -\sum_i y_i \cdot \log p_i = -\sum_i y_i \cdot x_i + \log \sum_j \exp(x_j),$$

which attains its minimum (zero loss) when $\boldsymbol{p} = \boldsymbol{y}$, indicating a perfect prediction. Throughout this paper, $\log(\cdot)$ denotes natural logarithm and we let $0 \log 0 = 0$. On the other hand, assigning a small probability p_i to the ground truth label ($y_i = 1$) can yield a large loss. As $\boldsymbol{y}$ is fixed, ℓ is equivalent to the Kullback-Leibler divergence (also known as the relative entropy) between $\boldsymbol{y}$ and $\boldsymbol{p}$ up to an additive constant.

Finally, for the optimization process, we take the partial derivative of the loss function in every coordinate and then perform gradient descent. This indicates the "correction" on the output values needed for a better model. In our example, the partial derivatives are

$$\begin{aligned}\nabla_i := \frac{\partial \ell}{\partial x_i} &= \frac{\partial}{\partial x_i}\Big(-\sum_k y_k \cdot x_k + \log \sum_j \exp(x_j)\Big) \\ &= -y_i + \frac{\exp(x_i)}{\sum_j \exp(x_j)} = -(y_i - p_i).\end{aligned}$$

When the loss reaches a local minimum, the partial derivatives are close to zero. Due to the derivatives of exp and log, softmax appears again in the expression of the gradient of ℓ. This is not true in general for arbitrary maps from real vectors to probability distributions, as we will see below.

3 A Softmax Replacement

Mohassel and Zhang [MZ17] suggested replacing softmax with Equation 2. It is easy to recognize the appeal of this function. The vector $\tilde{\boldsymbol{p}} = (\tilde{p}_0, \tilde{p}_1, \cdots)$ in Equation 2 is a probability distribution if there is at least one $\tilde{p}_i > 0$. Similar to Equation 1, it assigns the highest probability to the largest value of $\boldsymbol{x}$. Furthermore, piece-wise linear approximations have proven to be successful in other contexts such as logistic regression. Mohassel and Zhang have proposed replacing the sigmoid function by three-piece linear approximation, which has been shown to closely match the accuracy without the replacement on several datasets [MZ17, HHL+20].

The backpropagation implied by using the softmax replacement as an output probability distribution has not been spelled out in previous work. Using the softmax replacement with cross-entropy loss results in the following loss function (considering the general case when $\sum_i \mathsf{ReLU}(x_i) > 0$)

$$\begin{aligned}\tilde{\ell} &:= -\sum_i y_i \cdot \log \tilde{p}_i \\ &= -\sum_i y_i \cdot \log \frac{\mathsf{ReLU}(x_i)}{\sum_j \mathsf{ReLU}(x_j)} \\ &= -\sum_i y_i \cdot \log \mathsf{ReLU}(x_i) + \log\Big(\sum_j \mathsf{ReLU}(x_j)\Big).\end{aligned}$$

While ℓ is defined on $\mathbb{R}^L$ in the previous section, $\tilde{\ell}$ is only defined on the region

$$\Omega(\boldsymbol{y}) := \{\boldsymbol{x} : \boldsymbol{y}^\top \boldsymbol{x} > 0\} = \{\boldsymbol{x} : \forall i, x_i > 0 \text{ or } y_i = 0\} \subset \mathbb{R}^L,$$

which is a sub-region of $\mathbb{R}^L$ depending on $\boldsymbol{y}$. If $\boldsymbol{y}$ is a one-hot vector with $y_{i^\star} = 1$, then $\Omega(\boldsymbol{y})$ is the half-space $\{\boldsymbol{x} : x_{i^\star} > 0\}$.

For backpropagation, we take the partial derivative:

$$\begin{aligned}\frac{\partial \tilde{\ell}}{\partial x_i} &= -y_i \cdot \frac{[x_i > 0]}{x_i} + \frac{[x_i > 0]}{\sum_j \mathsf{ReLU}(x_j)} \\ &= -[x_i > 0] \cdot \left(\frac{y_i}{x_i} - \frac{1}{\sum_j \mathsf{ReLU}(x_j)} \right) \\ &= -\frac{[x_i > 0]}{x_i} \cdot (y_i - \tilde{p}_i)\,, \end{aligned} \tag{3}$$

where $[\cdot]$ denotes the Iverson bracket (1 if the condition is true, otherwise 0). The obvious issue is division by zero and numerical instability caused by the second term in the parentheses, that is, if all ReLU evaluations are zero or close to it. We found that this can be fixed by defining the gradient flow

$$\widetilde{\nabla}_i := \begin{cases} 0 & y_i = 0,\ x_i < \varepsilon \\ -1 & y_i = 1,\ x_i < \varepsilon \\ -\left(\frac{y_i}{x_i} - \frac{1}{\sum_j \mathsf{ReLU}(x_j)} \right) & \text{otherwise} \end{cases}$$

for some $\varepsilon \in (0, 1)$. One can easily check that, $\forall \boldsymbol{x} \in \Omega(\boldsymbol{y})$,

$$\sum_i \frac{\partial \tilde{\ell}}{\partial x_i} \widetilde{\nabla}_i \geq 0,$$

meaning that $\widetilde{\nabla}_i$ respects the gradient of $\tilde{\ell}$ and gives a non-decreasing direction of $\tilde{\ell}$. The reason to use a non-zero ε is to limit the scale of the partial derivative. Using softmax, ∇_i is guaranteed to be in $(-1, 1)$, and thus we aim to constrain $\widetilde{\nabla}_i$ in a similar range. If $x_i \geq \varepsilon$,

$$\left| \frac{y_i}{x_i} - \frac{1}{\sum_j \mathsf{ReLU}(x_j)} \right| = \frac{1}{x_i} |y_i - \tilde{p}_i| \leq \frac{1}{x_i} \leq \frac{1}{\varepsilon}.$$

Therefore $\widetilde{\nabla}_i \in [-\frac{1}{\varepsilon}, \frac{1}{\varepsilon}]$. In our experiments, we simply fix $\varepsilon = 0.1$ and show that this suffices for convergence, albeit with lower accuracy than softmax. Neither $\tilde{\ell}$ nor $\widetilde{\nabla}_i$ is a continuous function with respect to $\boldsymbol{x}$. Therefore, the learning process may suffer from instability.

Kanai et al. [KFYA18] have suggested mitigating the instability using the following probability distribution:

$$\bar{p}_i := \frac{\mathsf{ReLU}(x_i) + \varepsilon}{\sum_i (\mathsf{ReLU}(x_i) + \varepsilon)}$$

for $\varepsilon = 10^{-8}$. Computing the gradient as above, we get

$$\frac{\partial \bar{\ell}}{\partial x_i} = -\frac{[x_i > 0]}{x_i + \varepsilon} \cdot \left(y_i - \frac{\mathsf{ReLU}(x_i) + \varepsilon}{\sum_j (\mathsf{ReLU}(x_j) + \varepsilon)} \right),$$

whose absolute value is bounded by $1/(x_i + \varepsilon)$. This is similar to our $\widetilde{\nabla}_i$. Both implementations give a bounded gradient.

In the general case (see, e.g., alternatives of softmax [BCG+21]), the softmax function can be defined as

$$q_i := \frac{f(x_i)}{\sum_j f(x_j)},$$

where $f(x)$ is a positive function differentiable almost everywhere and non-decreasing on $\mathbb{R}$:

$$\forall x, f(x) > 0, \text{ and } f'(x) \geq 0 \text{ exists with exceptions.}$$

In the original softmax function, $f(x) = \exp(x)$. In the above $\bar{\boldsymbol{p}}$, $f(x) = \mathsf{ReLU}(x) + \varepsilon$. The partial derivative of the cross-entropy loss with respect to x_i has the general formula (the detailed deviations are straightforward and omitted for brevity)

$$-\frac{f'(x_i)}{f(x_i)}(y_i - q_i) = -\frac{\mathrm{d}}{\mathrm{d}x_i}\left[\log f(x_i)\right](y_i - q_i) = -(\log \circ f)'(x_i)(y_i - q_i).$$

Indeed, the logarithm is the integral of $1/x$ up to an additive constant, and the correctness follows via the chain rule. A sufficient condition for the gradient of the cross-entropy to be bounded is that the function $(\log \circ f)(x) := \log f(x)$ has a bounded gradient (equivalently, that $\log \circ f$ is Lipschitz). This condition holds when $f(x) = \exp(x)$ but may not hold for arbitrary f.

3.1 Replacing Softmax Directly in the Backpropagation

Following Mohassel and Zhang's proposal, several works [WTB+21, WGC19, CRS20, PSSY21, KPRS22] used the softmax replacement directly in backpropagation. This is to say, they implement gradient descent by manually modifying the gradient of ℓ with respect to x_i as

$$-(y_i - \tilde{p}_i) = -\left(y_i - \frac{\mathsf{ReLU}(x_i)}{\sum_i \mathsf{ReLU}(x_i)}\right).$$

Comparing this with the expression of $\frac{\partial \tilde{\ell}}{\partial x_i}$ in Equation 3, the non-negative coefficient $\frac{[x_i > 0]}{x_i}$ appearing in the true derivative has been dropped. As a result, the partial derivatives are bounded in magnitude. We have

$$\begin{aligned}\sum_i \frac{\partial \tilde{\ell}}{\partial x_i}\left[-(y_i - \tilde{p}_i)\right] &= \sum_i -\frac{[x_i > 0]}{x_i} \cdot (y_i - \tilde{p}_i) \cdot \left[-(y_i - \tilde{p}_i)\right] \geq 0 \\ &= \sum_i \frac{[x_i > 0]}{x_i}(y_i - \tilde{p}_i)^2 \geq 0.\end{aligned}$$

The inner product between the modified gradient and the original gradient of $\tilde{\ell}$ is non-negative. Therefore, reversing the modified gradient gives a descent direction of $\tilde{\ell}$.

Taking into account the special case when $\sum_j \mathsf{ReLU}(x_j) = 0$, we implement this approach as follows:

$$\widehat{\nabla}_i := \begin{cases} -y_i, & \text{if } \sum_j \mathsf{ReLU}(x_j) = 0 \\ -y_i + \frac{\mathsf{ReLU}(x_i)}{\sum_i \mathsf{ReLU}(x_i)}. & \text{otherwise} \end{cases}$$

We experimentally found that the issue is unlikely to arise for a random model, hence, it is not necessary to treat small values of $\sum_j \mathsf{ReLU}(x_j)$. This is because, under standard random initialization with zero-mean symmetric weights and biases as generated at the beginning of the training, the output distribution of the logits $\boldsymbol{x}$ is symmetrically distributed around zero. This variant is scale-invariant so that, for any $\lambda \neq 0$, replacing the softmax input $\boldsymbol{x}$ by $\lambda\boldsymbol{x}$ yields exactly the same $\widehat{\nabla}_i$. In other words, learning based on $\widehat{\nabla}_i$ loses sensitivity of the scale of the logits.

While this is less likely to require treatment of special cases, and it comes closer to the softmax backpropagation, the above works do not provide a formal justification in the form of a loss function. Nevertheless, a loss function or probability distribution is not necessary

Table 1: Comparison of softmax with the two replacement variants.

Variant	Gradient	Common	Differentiable	Known loss	Computation		
					$x < y$	x/y	$\exp(x)$
Softmax	∇_i	✔	✔	✔	✔	✔	✔
ReLU probability	$\widetilde{\nabla}_i$	✘	✘	✔	✔	✔	✘
ReLU gradient	$\widehat{\nabla}_i$	✘	✘	✘	✔	✔	✘

to measure the accuracy because that can simply be done by taking the maximum of the output values. We did so in our implementation. However, while we have managed to stabilize the accuracy in some cases, it remained considerably below the accuracy achieved with either softmax or the softmax replacement as the output probability distribution, that is, the solution at the beginning of Section 3. Table 1 summarizes the different variants.

4 Softmax in MPC

In this section, we present an implementation of softmax in arithmetic MPC that is largely independent of the actual protocol. Recall that many MPC protocols offer integer-like computation in the form of addition and multiplication modulo an integer, commonly a prime or a power of two. Reducing floating-point computation to these operations is possible [ABZS13] but expensive. Most works in privacy-preserving machine learning therefore use fixed-point representation, which has been introduced by Catrina and Saxena [CS10] in the context of MPC. It represents fractional numbers x as $\lfloor x \cdot 2^f \rceil$ for a precision parameter f where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. The concept is a special case of quantization used in machine learning [Guo18]. We will use fixed-point presentation for the rest of this section.

We start by summarizing the underlying arithmetic used in our algorithms. Commonly used multi-party computation schemes operate modulo a large enough integer, thus offering integer-like computation comparable to modern processors, where the modulus commonly is 2^{64}. This allows covering many schemes in a single framework [Kel20]. The following paragraphs refer to all arithmetic schemes in said work unless stated otherwise. We emphasise that these cover a wide range of settings and applications, from a limited number of parties with an honest majority and further assumption on malicious parties to an unlimited parties where only one of them is trusted, and all others' deviation from the protocol will be detected. The concrete choice in an application will depend on questions that go beyond this work, such as how much established trust there is between the participating entities.

The most important distinctions in MPC settings are honest vs. dishonest majority and semi-honest vs. malicious adversaries. Honest majority means that the trusted parties form a strict majority, for example, two out of three. Two-party computation is inherently dishonest-majority, a setting that is known to require more sophisticated cryptography to be realized [Kel20], such as homomorphic encryption or oblivious transfer. Oblivious transfer (OT) refers to a fundamental cryptographic protocol [Rab05]. MPC implementations generally use OT extension [Roy22] due to the cost of actual OT. Orthogonally, a semi-honest adversary means that even the corrupted (non-trusted) parties follow the protocol, whereas a malicious adversary is considered to deviate in any way. In the latter case, the most efficient protocols simply check for deviating behavior and abort when it is detected. We refer the reader to Lindell [Lin20] for a general introduction to MPC.

All the components below are to be understood in the arithmetic black box model, where the parties instruct the black box to operate on values it holds. Additional operations are based on existing ones and considered to be added to the black box for simplicity.

Addition Integer addition is possible without interaction in all relevant schemes, and fixed-point addition corresponds to integer addition.

Multiplication Integer multiplication is possible with interaction in all relevant schemes. Multiplying two integers representing fixed-point numbers with precision f results in an integer representation with precision $2f$ as $x \cdot 2^f \cdot y \cdot 2^f = xy \cdot 2^{2f}$. Integer division by 2^f is necessary to revert to precision f. Catrina and Saxena proposed a probabilistic method particularly suited for MPC, which rounds up or down with probability according to the distance. If $f = 2$, for example, $x = 1 \mod 4$ would be rounded down with probability 0.75. This method is used by some in privacy-preserving machine learning, including Mohassel and Zhang [MZ17].

Another possibility is nearest rounding, which can be implemented by adding 2^{f-1} before applying truncation or right-shift by f bits. A general protocol is available due to Catrina and de Hoogh [CdH10], and Dalskov et al. [DEK20, DEK21] have proposed optimized protocols for both truncation options with honest-majority MPC. Furthermore, Escudero et al. [EGK+20] have explored the benefits of mixed-circuit computation for truncation.

Mixed-circuit computation Secret sharing requires the shares to be elements of a group or ring. Most commonly, this takes the form of integers modulo a number. If this number is two, we speak of binary computation, otherwise of arithmetic computation. The latter facilitates integer computation, as does computation modulo 2^{64} on modern CPUs, whereas the former facilitates computation of binary circuits. Demmler et al. [DSZ15] started a line of works exploring the benefit of combining the two approaches. Rotaru and Wood [RW19] introduced so-called daBits, which allow doing this in any scheme covered here. More recently, Escudero et al. [EGK+20] proposed so-called edaBits for mixed-circuit computation. These are random values that are shared in both domains, which allows for the following approach to conversion: mask a secret value in one domain by addition, open the result, input the masked public value in the other domain, and subtract the mask again. Escudero et al. have used edaBits to construct several specific computations, such as comparison. Using edaBits is possible in any security model, but some honest-majority protocols enable more efficient constructions via local share conversion [DSZ15, MR18, ABF+18].

All the above can be used for A2B and B2A in Section 4.1. The concrete MPC scheme informs which technique is most efficient. Demmler et al.'s only works in the case of semi-honest two-party computation, and [MR18, ABF+18] only work for semi-honest three-party computation with one corruption. In all other cases, Escudero et al. [EGK+20] show that using edaBits improves on simple daBits in many cases. We refer to Keller and Sun [KS22] as well as the documentation of MP-SPDZ[1] for more details.

Comparison There is a considerable line of works regarding comparison in arithmetic MPC. Damgård et al. [DFK+06] proposed a method that allows any input, while Catrina and de Hoogh [CdH10] work with inputs in a restricted range. Further works [MR18, EGK+20] improve the efficiency by using mixed circuits.

Fixed-point division Catrina and Saxena [CS10] have applied Goldschmidt's iteration [Gol64] to the MPC setting. They obtain an initial approximation by finding the first 1 in a bit decomposition of the input value. It is straightforward to use a mixed-circuit technique for this.

[1] https://mp-spdz.readthedocs.io/en/latest/non-linear.html#mixed-circuit-computation

Algorithm 1 Exponentiation with base two [KS22]

Input: Secret share $\langle x \rangle$ with precision f where $x < k - f - 1$
Output: $\langle 2^x \rangle$ with precision f and total bit length k

1: $[x_0], \cdots, [x_{k-1}] \leftarrow \mathsf{A2B}(\langle x \rangle)$ ▷ bit decomposition
2: $[z] \leftarrow \sum_{i=0}^{k-1} 2^{i-f}[x_i] < -(k-f-1)$ ▷ binary comparison with fixed-representation
3: $\ell \leftarrow \lceil \log_2(k-f) \rceil$
4: **for** $j = f, \ldots, f + \ell - 1$ **do**
5: $\quad [\![x_j]\!] \leftarrow \mathsf{B2A}([x_j])$
6: **end for**
7: $[\![d]\!] \leftarrow \prod_{j=0}^{\ell-1}([\![x_{f+j}]\!] \cdot 2^{2^j} + 1 - [\![x_{f+j}]\!])$ ▷ integer exponentiation
8: $[\![r]\!] \leftarrow \mathsf{B2A}([x_0], \ldots, [x_{f-1}])$ ▷ fractional component
9: $\langle u \rangle \leftarrow \mathsf{Approx}_{2^*}(\langle r \rangle)$ ▷ polynomial approximation
10: $\langle g \rangle \leftarrow \langle u \rangle \cdot [\![d]\!]$
11: $\langle g' \rangle \leftarrow \mathsf{Round}(\langle g \rangle, f + 2^\ell, 2^\ell)$ ▷ result for negative input
12: $\langle h \rangle \leftarrow \mathsf{B2A}([x_{k-1}]) \cdot (\langle g' \rangle - \langle g \rangle) + \langle g \rangle$ ▷ correct for sign
13: **return** $(1 - \mathsf{B2A}([z])) \cdot \langle h \rangle$ ▷ output 0 if input is too small

4.1 Exponentiation

Aly and Smart [AS19] presented a protocol for exponentiation that roughly works as follows:

1. Reduces to exponentiation with base two
2. Split the exponent into the integer and fractional parts
3. Compute the integer exponentiation precisely and approximate the fractional exponentiation
4. Multiply the two results

Keller and Sun [KS22] have subsequently improved the accuracy and performance of this approach. Algorithm 1 shows their resulting algorithm. It only works for base two, but the equality $B^e = 2^{e \log_2 B}$ can be used to reduce any exponentiation to exponentation with base two. It uses the Taylor approximation of 2^x for $x \in [0, 1]$. Beyond that, the algorithm uses standard components as described earlier.

Throughout this section, we use $[\![\cdot]\!]$ to indicate integer sharing, $\langle \cdot \rangle$ to indicate fixed-point sharing, and $[\cdot]$ for binary secret sharing.

Now, we generalize the two-party approach by Rathee et al. [RRG+21]. This protocol uses look-up tables on parts of the input before multiplying the results. Algorithm 2 shows the high level. The main difference to Keller and Sun is using look-up tables, whereas the main difference to Rathee et al. is generalization beyond the two-party setting.

Note that the algorithm requires an upper limit for the input and outputs zero for negative inputs. In the next section, we will show that this suffices for softmax computation. In every round of the loop, a look-up table on C bits is computed using the integer representation of the 2^C possible results represented by the bits $x_{iC}, \ldots, x_{iC+C-1}$. The correctness stems from the fact that $B^{x+y} = B^x \cdot B^y$. The chunks sum up to the input, hence one can multiply the chunk-wise exponentiation to get the final result. Furthermore, step 5 ensures that the output is zero for negative inputs by using the sign bit. The security is inherited from the building blocks because they are used in a black-box manner common to multi-party computation.

Algorithm 3 shows the look-up table algorithm. All the communication takes place in the sub-algorithm 4, which converts binary to unary representation. For example, the binary representation of five is 101, and the unary representation is $0000010 \ldots 0$. This

Algorithm 2 Exponentiation based on look-up tables

Input:

- Base B
- Secret share $\langle x \rangle$ with precision f where $-(k-f-1) < x < M$ such that $B^M < 2^{k-f-1}$
- Chunk size C

Output: $\langle B^x \rangle$ with precision f and total bit length k if $x > 0$, $\langle 0 \rangle$ otherwise

1: $[x_0], \cdots, [x_{k-1}] \leftarrow \mathsf{A2B}(\langle x \rangle)$ ▷ bit decomposition
2: **for** $i = 0, \ldots, \lceil \log_2(M)/C \rceil$ **do**
3: $[y_i] \leftarrow \mathsf{LUT}((x_{iC}, \ldots, x_{iC+C-1}), (\lfloor B^{0 \cdot 2^{iC-f}} \cdot 2^f \rceil, \ldots, \lfloor B^{(2^C-1) \cdot 2^{iC-f}} \cdot 2^f \rceil))$
4: **end for**
5: $[y_0] \leftarrow [y_0] \cdot (1 - [x_{k-1}])$ ▷ element-vector AND
6: **return** $\prod_{i=0}^{\lceil \log_2(M)/C \rceil} \mathsf{B2A}([y_i])$ ▷ multiply as fixed-point values

Algorithm 3 Look-up table LUT

Input:

- Shared bits $[x_0], \ldots, [x_{C-1}]$
- Possible clear-text output $L_0, \ldots, L_{2^C-1}$

Output: $[L_{\sum_{i=0}^{C-1} x_i \cdot 2^i}]$

1: $[X_0], \ldots, [X_{2^C-1}] \leftarrow \mathsf{Demux}(x_0, \ldots, x_{C-1})$
2: **return** $\bigoplus_{i=0}^{2^C-1} [X_i] \cdot L_i$ ▷ $\cdot$ denotes element-vector AND

is commonly used in MPC for oblivious selecting an element from a list, going back to at least Launchbury et al. [LADM14]. Computing the dot product between the unary representation of n and a list of length at least n will produce the n-th element of the list. Note that multiplication in the binary domain is equivalent to AND, and that $\oplus$ denotes XOR.

Benchmarks We have implemented our algorithm in MP-SPDZ [Kel20]. Table 2 shows the communication cost in comparison to Keller and Sun [KS22] for several underlying protocols and $f = 16, k = 31$. It shows that our method provides an improvement of a factor up to 4.2. The table also shows the costs per ReLU operation. While they are considerably lower, the overall benchmarks in Table 3 highlight the limited influence. Note that our cost for two-party computation higher than [RRG+21], who quote 17 kbit for power-of-two-modulus computation and 39 kbit for prime-modulus computation. Our two-party computation uses the latter for compatibility with homomorphic encryption. We found that we can match their result by increasing the parameter for SoftSpokenOT [Roy22] to ten. In MP-SPDZ, the default is two. The parameter represents a computation-communication tradeoff, and increasing the parameter to ten increases the computation time six-fold while reducing communication by less than three-fold.

4.2 Softmax

Recall the definition of softmax in Equation 1. Using the exponential function with fixed-point representation limits the input to $k - f - 1$ where k is the total number of bits and f is the precision parameter. A common choice would be $k = 31$ and $f = 16$ (allowing division with $\mathbb{Z}_{2^{64}}$ as the computation domain), allowing only inputs up to 14. This is clearly very restrictive. However, the division in softmax makes it possible to normalize

Algorithm 4 $([d_0], \ldots, [d_{2^l-1}]) \leftarrow \mathsf{Demux}([a_0], \ldots, [a_{l-1}])$

Input: Bit decomposition of $[a]$
Output: Bit vector $([d_0], \ldots, [d_{2^l-1}])$ such that $d_a = 1$ and $d_i = 0$ for all $i \neq a$

1: **if** $l = 1$ **then**
2: **return** $(1 - [a_0], [a_0])$
3: **end if**
4: $([b_0], \ldots, [b_{2^{\lfloor l/2 \rfloor}-1}] \leftarrow \mathsf{Demux}([a_0], \ldots, [a_{\lfloor l/2 \rfloor}-1])$
5: $([c_0], \ldots, [c_{2^{l-\lfloor l/2 \rfloor}-1}] \leftarrow \mathsf{Demux}([a_{\lfloor l/2 \rfloor}], \ldots, [a_{l-1}])$
6: **for** $i = 0$ to $2^l - 1$ **do**
7: $[d_i] = [b_{i \bmod 2^{\lfloor l/2 \rfloor}}] \cdot [c_{i/2^{\lfloor l/2 \rfloor}}]$
8: **end for**
9: **return** $([d_0], \ldots, [d_{2^l-1}])$

Table 2: Communication cost in kbit per exponentiation, ReLU, and multiplication.

Protocol	# parties	# corruptions	Malicious	Our exp.	[KS22] exp.	ReLU	Mult.
[KS22]	2	1	✘	94	113	21.9	3.6
[KS22]	3	1	✘	6	15	0.7	0.2
[DEK21]	3	1	✔	85	323	6.3	0.6
[DEK21]	4	1	✔	15	27	1.8	0.4
[GLO+21]	10	4	✘	3,003	12,736	526.0	2.6

the inputs. For any $X \in \mathbb{R}$,

$$\frac{\exp(x_i)}{\sum_j \exp(x_j)} = \frac{\exp(x_i - X)}{\sum_j \exp(x_j - X)}.$$

Previous work [TKTW21, KS22] used $X = \max_i x_i$, which ensures that all exponents are at most zero, limiting the range of $\exp(x_i)$ to $[0, 1]$. We propose another method to ensure that the most relevant exponents will be positive while keeping all $\sum_i \exp(x_i - X)$ within the range of our fixed-point representation. More concretely, we let

$$X^\star := \max_i x_i - \log 2^{k-f-1} + \log L.$$

It follows that

$$\begin{aligned}\sum_{i=0}^{L-1} \exp(x_i - X^\star) &= \sum_{i=0}^{L-1} \exp\left((x_i - \max_i x_i) + \log 2^{k-f-1} - \log L\right) \\ &\leq L \cdot \exp(\log 2^{k-f-1} - \log L) \\ &= 2^{k-f-1},\end{aligned}$$

which is below the maximum value for our representation. In the context of MPC, $X^\star$ can be computed using a tree of the following operation:

$$\max(x, y) := x + (y - x) \cdot (x \overset{?}{<} y),$$

where the last component is the usual comparison operation producing a secret bit.

Recall that Algorithm 2 outputs zero for negative values instead of the correct value in $[0, 1]$. In the next section, we will show experimentally that this does not affect the accuracy of the training. Consider x_i such that $x_i - X^\star < 0$. It follows that

$$x_i - X^\star = x_i - \max_j x_j + \log 2^{k-f-1} - \log L \approx x_i - \max_j x_j + 7.4 < 0$$

```
layers.Flatten(),
layers.Dense(128, activation='relu'),
layers.Dense(128, activation='relu'),
layers.Dense(10, activation='softmax')
```

Figure 1: Model with three dense layers.

for $k = 31$, $f = 16$ and $L = 10$, and therefore

$$x_i - \max_j x_j < -7.4.$$

Such a value of x_i yielding an incorrect $\exp(x_i - X^\star)$ is considerably smaller than $\max_j x_j$. We have

$$p_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \leq \frac{\exp(x_i)}{\exp(\max_j x_j)} < \exp(-7.4) \approx 6 \times 10^{-4}.$$

Our Algorithm 2 rounds the above p_i to zero, which is acceptable based on its tiny magnitude. As another example, for binary classification, $L = 2$ and any $p_i < \exp(-9.0) \approx 1 \times 10^{-4}$ will be rounded to zero.

Intuitively, the idea of softmax is to produce a differentiable maximum selection. Hence, values that are relatively far from the maximum are much less relevant to the outcome. Consider the example input

$$(5, 5, 0, \ldots, 0) \in \mathbb{R}^{10}.$$

The softmax thereof is

$$(0.48687777, 0.48687777, 0.00328056, \ldots, 0.00328056).$$

Now consider a slight change in one of the maximum entries to

$$(5, 4.9, 0, \ldots, 0),$$

which results in a softmax of

$$(0.51053202, 0.46194848, 0.00343994, \cdots, 0.00343994).$$

The change of the probability vector measured by the max norm is more than 0.024. In comparison, the softmax of

$$(5, 5, -0.1, 0, \ldots, 0)$$

is

$$(0.48702982, 0.48702982, 0.0029693, 0.00328158, \ldots, 0.00328158).$$

The change of the probability vector is less than 0.0004. This illustrates the lower impact of inaccurate exponentiation of smaller values when evaluating the softmax function.

5 Experiments

We implemented training for the MNIST dataset [LBBH98] in MP-SPDZ [Kel20] with one to three dense layers. All but the last layer consist of 128 ReLU units [GBB11]. Figure 1 shows the three-layer model as Keras [C+15] code. MNIST is a widely used dataset, which consists of 70,000 gray-scale pictures of dimension 28×28 depicting hand-written digits 0–9. Our implementation is based on Keller and Sun's [KS22], where we refer the reader for details.

Table 3: Time and accuracy for various models and parameters. "$\perp$" stands for divergence even with the smallest possible learning rate. "Rounding" denotes the rounding after multiplication in fixed-point representation (probabilistic or nearest), and "ReLU probability" and "ReLU gradient" denote using the softmax replacement for output probability and the gradient, respectively. Bold face indicates the best result.

No. layers	Rounding	Backpropagation	s/ep.	GB/ep.	Accuracy after n epochs			
					$n = 5$	$n = 10$	$n = 15$	$n = 20$
1	Prob.	Our softmax	8.6	0.5	91.6	92.0	92.2	92.2
		Softmax [KS22]	8.5	0.5	91.7	92.1	92.2	92.3
		ReLU probability	5.1	0.3	89.6	90.0	89.9	90.7
		ReLU gradient	4.7	0.2	86.6	86.6	86.6	86.6
	Nearest	Softmax	14.0	1.4	91.8	92.2	92.3	92.4
		Softmax [KS22]	16.2	1.9	91.8	92.2	92.4	92.5
		ReLU probability	9.7	1.3	90.5	90.3	89.2	90.1
		ReLU gradient	9.6	0.9	86.6	86.6	86.5	86.6
2	Prob.	Our softmax	23.3	2.4	95.9	97.0	97.4	97.6
		Softmax [KS22]	23.8	2.4	95.8	96.9	97.4	97.5
		ReLU probability	20.1	2.3	93.3	94.1	94.7	94.9
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	Nearest	Softmax	39.7	11.5	96.2	97.2	97.4	97.5
		Softmax [KS22]	42.9	12.1	96.2	97.1	97.3	97.5
		ReLU probability	34.8	11.4	92.7	91.6	89.9	88.0
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
3	Prob.	Our softmax	28.4	3.2	96.7	97.4	**97.7**	**97.8**
		Softmax [KS22]	29.0	3.2	96.8	**97.5**	**97.7**	97.7
		ReLU probability	24.4	3.1	91.5	92.8	93.3	93.3
		ReLU gradient	24.2	2.9	82.6	87.0	88.4	88.8
	Nearest	Our softmax	49.8	14.4	**97.0**	97.4	97.5	97.5
		Softmax [KS22]	53.0	15.0	96.9	**97.5**	97.5	97.4
		ReLU probability	44.6	14.3	93.1	93.5	92.9	90.5
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 4: Standard deviation for accuracy results.

No. layers	Rounding	Backpropagation	Accuracy deviation after n epochs			
			$n = 5$	$n = 10$	$n = 15$	$n = 20$
1	Prob.	Our softmax	0.1	0.1	0.1	0.1
		Softmax [KS22]	0.1	0.1	0.1	0.1
		ReLU probability	0.9	0.5	0.5	0.5
		ReLU gradient	0.1	0.1	0.0	0.0
	Nearest	Softmax	0.1	0.1	0.1	0.1
		Softmax [KS22]	0.1	0.1	0.1	0.1
		ReLU probability	0.4	0.6	1.5	0.5
		ReLU gradient	0.1	0.0	0.1	0.0
2	Prob.	Our softmax	0.2	0.1	0.1	0.1
		Softmax [KS22]	0.1	0.1	0.2	0.1
		ReLU probability	0.4	0.3	0.2	0.3
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$
	Nearest	Softmax	0.1	0.1	0.1	0.1
		Softmax [KS22]	0.1	0.1	0.1	0.1
		ReLU probability	1.4	2.9	5.3	4.8
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$
3	Prob.	Our softmax	0.1	0.0	0.0	0.0
		Softmax [KS22]	0.1	0.1	0.1	0.2
		ReLU probability	1.9	1.6	1.7	1.8
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$
	Nearest	Our softmax	0.1	0.1	0.1	0.0
		Softmax [KS22]	0.1	0.1	0.1	0.1
		ReLU probability	1.1	1.6	2.2	4.8
		ReLU gradient	$\perp$	$\perp$	$\perp$	$\perp$

Table 3 lists our timings and accuracy results for one run of each variant with honest-majority semi-honest three-party computation on colocated AWS c5.9xlarge instances. They feature 36 virtual CPUs, 72 GB of memory, and network bandwidth of 10 Gbps. Our code is available as a Docker container for reproduction.[2]

Our results show that the ReLU-based softmax replacement only improves the running time per epoch by less than 20 per cent for two layers or more, while it considerably deteriorates the accuracy for any number of layers. For two or three layers, the testing error for the real softmax is generally less than half compared to the MPC version. For example, the best error is 2.2 percent with the original softmax (three layers and 20 epochs), but the best error with an MPC-friendly variant is 6.7 percent. Table 4 also shows that the MPC-friendly variants exhibit much variance over at least three runs.

If measuring the time it takes until a certain accuracy is reached, softmax always produces the best results. Furthermore, using the ReLU-based replacement directly in the backpropagation is unlikely to produce convergence with more than one layer, and where it does, the results are far inferior. Table 3 also shows that our exponentiation does not decrease accuracy compared to the exponentiation by Keller and Sun [KS22].

Further notable is the fact that three layers only marginally improve the accuracy for the reported number of epochs. We found that 98 per cent are achieved after 50 epochs with both two and three layers. The same occurs in plaintext training, where we found that a three-layer model would not improve over two layers. This was done using the TensorFlow MNIST tutorial [Bra] by running it as is (two dense layers) and duplicating the first dense layer. The three-layer model was used by Mohassel and Zhang [MZ17] and later dubbed Network A by Wagh et al. [WGC19].

The more recent work of Wagh et al. [WTB+21] reports a timing of 0.17 hours for 15

[2] https://github.com/csiro-mlai/deep-mpc

epochs of training, which corresponds to 41 seconds per epoch. They improve on previous works [MR18, WGC19] in the same security model, which shows that our implementation is competitive even when using softmax.

We have also run two-party training for one dense layer with probabilistic rounding. We found that one epoch takes 943, 942, 922, and 894 seconds with [KS22] softmax, our softmax, ReLU probability, and ReLU gradient, respectively. ReLU gradient sometimes diverges at deeper networks, as it does not correspond to minimizing a known loss function, and its convergence is not guaranteed. As discussed previously, $\widehat{\nabla}_i$ is not sensitive to the input scale. Optimization is destabilized, with the effect amplified as the gradient is propagated through multiple layers. The relative benefit of the softmax replacements thus is even smaller, which can be explained by the fact that the cost of matrix multiplication in our three-party computation only depends on the output size, whereas the optimized two-party computation by Keller and Sun (based on the diagonal packing by Halevi and Shoup [HS14]) depends on the output and input sizes. This makes the linear operations more expensive in relation to the non-linear operations.

6 Conclusion

We conclude that the softmax replacement by Mohassel and Zhang [MZ17] is of limited use. For inference, finding the index with the maximum value in a vector (argmax) is often enough. For training, on the other hand, the replacement proves to deteriorate accuracy and slow down convergence to such an extent that softmax is more efficient in reaching a target accuracy. Our conclusion is only based on empirical results for a relatively simple network. However, we consider the increasing instability with the number of layers strong evidence against the softmax replacement being more useful in more complex networks. Furthermore, we do not see a path to strengthening the analysis of the MPC-friendly variants. The original softmax has a strong theoretical underpinning in its universal differentiability, which the newer variants do not have.

References

[ABF+18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the SPDZ compiler for other protocols. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 880–895. ACM Press, October 2018. doi:10.1145/3243734.3243854.

[ABZS13] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS 2013*. The Internet Society, February 2013.

[AS19] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 2019*, volume 11464 of *LNCS*, pages 509–529. Springer, Cham, June 2019. doi:10.1007/978-3-030-21568-2_25.

[BCG+21] Kunal Banerjee, Vishak Prasad C, Rishi Raj Gupta, Karthik Vyas, Anushree H, and Biswajit Mishra. Exploring alternatives to softmax function. In *Proceedings of the 2nd International Conference on Deep Learning Theory and Applications (DeLTA 2021)*, pages 81–86, 2021.

[Bra] Google Brain. Training a neural network on MNIST with Keras. https://www.tensorflow.org/datasets/keras_example.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[CdH10] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Berlin, Heidelberg, September 2010. doi:10.1007/978-3-642-15317-4_13.

[CRS20] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS 2020*. The Internet Society, February 2020. doi:10.14722/ndss.2020.23005.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Berlin, Heidelberg, January 2010. doi:10.1007/978-3-642-14577-3_6.

[DEK20] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, 2020(4):355–375, October 2020. doi:10.2478/popets-2020-0077.

[DEK21] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2183–2200. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Berlin, Heidelberg, March 2006. doi:10.1007/11681878_15.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015. doi:10.14722/ndss.2015.23113.

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020. doi:10.1007/978-3-030-56880-1_29.

[Fog25] Agner Fog, 2025. Accessed February 2026. URL: https://agner.org/optimize/instruction_tables.pdf.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *AISTATS*, volume 15 of *PMLR*, pages 315–323, 2011.

[GLO+21] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 244–274, Virtual Event, August 2021. Springer, Cham. doi:10.1007/978-3-030-84245-1_9.

[Gol64] Robert E. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, 1964.

[Guo18] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[HHL+20] Cheng Hong, Zhicong Huang, Wen-jie Lu, Hunter Qu, Li Ma, Morten Dahl, and Jason Mancuso. Privacy-preserving collaborative machine learning on genomic data using tensorflow. *arXiv preprint arXiv:2002.04344*, 2020.

[HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 554–571. Springer, Berlin, Heidelberg, August 2014. doi:10.1007/978-3-662-44371-2_31.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020. doi:10.1145/3372297.3417872.

[KFYA18] Sekitoshi Kanai, Yasuhiro Fujiwara, Yuki Yamanaka, and Shuichi Adachi. Sigsoftmax: Reanalysis of the softmax bottleneck, 2018. arXiv:1805.10829.

[KPRS22] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4PC for secure training and inference. In *NDSS 2022*. The Internet Society, April 2022.

[KS22] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 10912–10938. PMLR, 17–23 Jul 2022.

[LADM14] John Launchbury, Dave Archer, Thomas DuBuisson, and Eric Mertens. Application-scale secure multiparty computation. In Zhong Shao, editor, *Programming Languages and Systems*, pages 8–26, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[Lin20] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020. doi:10.1145/3387108.

[MA16] André F. T. Martins and Ramón F. Astudillo. From softmax to sparsemax: a sparse model of attention and multi-label classification. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pages 1614–1623, 2016.

[MKH19] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. When does label smoothing help? In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018. doi:10.1145/3243734.3243760.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017. doi:10.1109/SP.2017.12.

[PNM19] Ben Peters, Vlad Niculae, and André FT Martins. Sparse sequence-to-sequence models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1504–1519, 2019.

[PSSY21] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2165–2182. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/patra.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. URL: https://eprint.iacr.org/2005/187.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Cham, August 2022. doi:10.1007/978-3-031-15802-5_23.

[RRG+21] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SiRnn: A math library for secure RNN inference. In *2021 IEEE Symposium on Security and Privacy*, pages 1003–1020. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00086.

[RW19] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Cham, December 2019. doi:10.1007/978-3-030-35423-7_12.

[TKTW21] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *2021 IEEE Symposium on Security and Privacy*, pages 1021–1038. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00098.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019. doi:10.2478/popets-2019-0035.

[WTB+21] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):188–208, January 2021. doi:10.2478/popets-2021-0011.