Privacy Preserving Neural Network: Secure Framework in 3-Party Setting

Capstone Project

Kuber Shahi
Dept. of Computer Science,
Ashoka University

Advisors:

Dr. Mahavir Jhawar
Dept. of Computer Science,
Ashoka University

Dr. Debayan Gupta
Dept. of Computer Science,
Ashoka University

December 29, 2021

Abstract

The aim of this work is to study privacy privacy machine learning (PPML) for neural network in three-party (3PC) setting. First, I have studied and implemented a Neural Network (NN) to understand its building blocks and derivation of its update functions through backpropagation algorithm for different activation and cost functions. For this work, I have primarily focused on an NN with ReLU in the hidden layers and softmax in the output layer that solves a multi-class classification problem. Second, I have studied privacy preserving neural network frameworks: SecureML[7] in two-party setting and SecureNN[11] in three-party setting. In this work, I have extensively described the mapping and truncation techniques from SecureML, and the building blocks, namely secure multiplication, comparison and division, from SecureNN. For my understanding, I have implemented[10] some of the functionalities from SecureML and SecureNN.

Contents

1. Introduction

"With data being the new science and privacy our new concern, privacy preserving machine learning is the place where insight meets privacy."

With the advancement in computing power, it comes as no surprise that machine learning (ML), a subset of AI (Artificial Intelligence), has gained mainstream popularity for building effective tools that enable computers to tackle tasks that have, until now, only been carried out by humans. From online companies deploying it to recommend their products and enhance user experience to healthcare institutions utilizing it for better and faster diagnosis of abnormalities, ML-based systems and solutions have found widespread applications in several areas such as banking, business, healthcare, technology, and so on. The key thing to note is that the robustness and accuracy of such systems and solutions are dependent on the availability of an enormous amount of data from various sources. The accumulation and sharing of data from and to various sources are not possible because of various reasons such as data regulations and guidelines, the data shared being sensitive or proprietary information, high cost for collection and maintenance of data. Considering the attraction and potential for ML-based solutions and the concurrent need to maintain the privacy of data being shared for training, the field of privacy preserving machine learning (PPML) has recently become a dominant area of research.

To formulate the problem more systematically, let's consider a scenario where N hospitals have agreed to share their patients' liver data to train an ML model that could help them quickly and accurately detect liver cancers in their future patients. Despite their willingness to do so, they can't directly share their patient's data as the privacy of the patients' data is protected by data-protection laws and regulations. But if they can guarantee that the privacy of the shared patients' data is maintained, then they can share the data among themselves and collaboratively train their ML model. So, now the problem boils down to N hospitals wanting to collaboratively and securely train an ML model on their shared data. A generic solution to this problem is given by Secure multi-party computation (MPC). Using MPC-employed frameworks [5, 6], the N hospitals can collaboratively perform calculations on their shared data to train their ML Model without revealing the identity of their inputs. These MPC-employed frameworks work over low-level circuits (either arithmetic or boolean) which works faster and efficiently for general tasks involving smaller inputs. But for complex tasks such as training ML models which are already compute-intensive, this leads to highly inefficient protocols that have high computational and time complexity which is undesirable for most companies in terms of cost and urgency for time-critical tasks.

Considering the high computational need of MPC-employed ML training schemes, it is more economical and suitable for the N parties to outsource the task to powerful and specialized systems such as servers. In this setting, N parties secretly share (2.2.1) their data to a number of these servers and the servers collaboratively train the ML model on the shared data. After the ML model is trained, the servers output the model back to the N parties or to another server to which all the N parties have access. However, throughout the process of sourcing and training, the privacy and security of the shared data must be ensured which is why this setting is formally known as Secure Outsourced Computation (SOC). Most state-of-the-art PPML frameworks and protocols today use a SOC setting comprising of two to four servers. Depending on the number of servers, supposedly n, being used in the SOC setting, the PPML framework/protocol is known as an n-Party Computation (n-PC) framework/protocol.

For my capstone project, I will be focusing on frameworks developed for Neural Network training and inference in 2PC and 3PC setting.

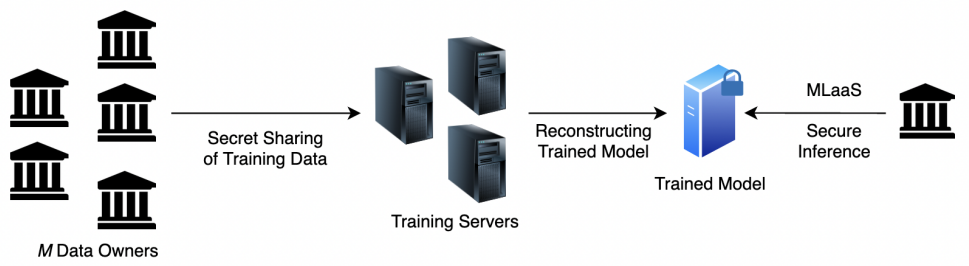

Figure 1: PPML Architecture

Report Outline: First of all, I have outlined and explained the background in machine learning and cryptography required to understand the SecureNN[11] paper in section 2. I have implemented a scratch version of plain NN, the details of which is mentioned in section 2.1.5. Then, after providing a brief overview of the SecureNN paper in section 3.1, I have illustrated the supporting and main protocols in section 3.2 and section 3.3 respectively. I have implemented some of the supporting protocols, the details of which are mentioned in the same section. At the end, in section 3.4, I have described the integration of the main protocols to build the final privacy preserving neural network (PPNN).

2. Preliminaries

2.1. Machine Learning (ML)

2.1.1. Linear Regression

Given n training data samples (x_i, y_i) where x_i has d features, $x_i \in \mathbb{R}^d$, with $y_i \in \mathbb{R}$ as it corresponding target labels, linear regression is a statistical method to approximate a function $h(x_i) \approx y_i$ with an assumption that the relationship between x_i and y_i is linear. With that assumption the function h can be represented as:

$$\hat{y}_i = h(x_i) = x_i \cdot w = \sum_{j=1}^{d} x_{ij} w_j \tag{1}$$

where $\hat{y}_i$ is the predicted output of x_i, x_{ij} is the j-th value of in vector x_i and w_j is its corresponding adjustable parameter in vector w, and $\cdot$ is the inner product.

2.1.2. Stochastic Gradient Descent (SGD)

To learn adjustable weight vector w, a loss function $J(w)$ is defined where w is given by $\text{argmin}_w J(w)$. In linear regression, mean-squared error (MSE), $J(w) = \frac{1}{n}\sum_i^n (\hat{y}_i - y_i)^2$, is a commonly used loss function. The loss function is defined in a such a way that it on one hand it measures the error between the predicted output $(\hat{y}_i)$ and the target (y_i), and on the other, it is convex making it easier to find the global minima.

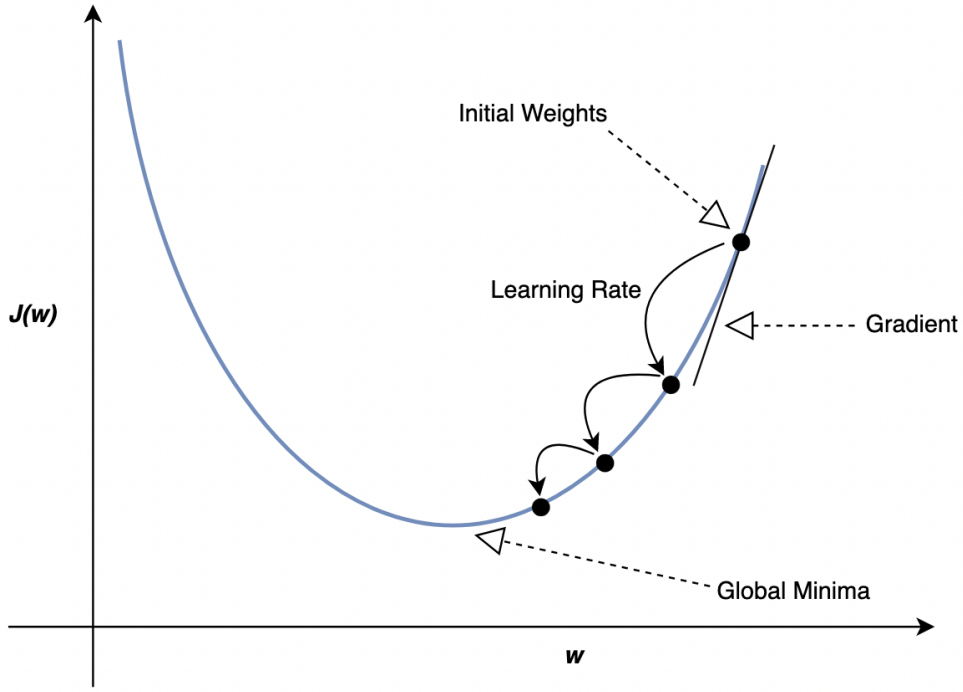

Figure 2: Stochastic Gradient Descent (SGD)

To find the global minima, SGD is used. SGD iteratively converges to the global minima by moving in the negative direction of the gradient of $J(w)$ as shown in the Figure 2 and updates the w at each iteration as:

$$w_j(new) := w_j(old) - \alpha \frac{\partial J_i(w)}{\partial w_j} \tag{2}$$

where α is the learning rate which defines how the magnitude of the step towards the minimum in each step and $\frac{\partial J_i(w)}{\partial w_j}$ is the partial derivative of J with respect to the weight, w_j, being updated. The method of calculated the partial derivative is called backpropagation. In addition, SGD can be generalized to other models such as logistic regression and neural networks which makes it the most common approach to train models in machine learning.

Weights can be updated through two ways: first, updating it each time after passing through a data point which is called online training, and second, updating it at the end after passing through all the data points which is called batch training. One pass through the entire dataset is called an epoch and typically the model is trained for more than two/three epochs. The number of epochs to train the model depends on the nature of the model and is based on heuristics until a good convergence to the global minima is found.
For the large dataset, the entire dataset can be too large to process due to memory constraints making batch learning difficult whereas the error are really noisy with single data point in online training. Therefore, in practice, a small batch of data points are selected randomly to train which is called mini-batch training and w is updated by averaging the partial derivatives over mini-batches for each iteration. This changes the update function for w as:

$$w(new) := w(old) - \frac{\alpha}{|B|} \frac{\partial J_B(w)}{\partial w} \tag{3}$$

where B is the mini-batch size and J_B is the error calculated over the mini-batch.

2.1.3. Logistic Regression

In linear regression, the target label of the input data points are continuous but not all problems have continuous output some of them have binary output as well. For instance, consider the problem of whether to buy the house or not given certain d features of house. Such problems are called binary classification function where the predicted output is either 0 or 1. To bound the predicted output $\hat{y}$, an activation function is applied on top of the inner product and the relationship is expressed as:

$$\hat{y}_i = h(x_i) = f(x_i \cdot w) = f\left(\sum_{j=1}^{d} x_{ij} w_j\right) \tag{4}$$

In logistic regression, the activation function used is logistic function, $f(x) = \frac{1}{1+e^{-x}}$ which converges to 0 and 1 at its two tails. With addition of logistic function, binary cross entropy loss function, $J(w) = -\sum_i^n y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$, is used to keep the loss function still convex so that SGD can be applied on it. The update function for the logistic regression remains the same.

2.1.4. Activation and Loss Functions

Over time, numerous activation and loss functions have been defined to handle and solve different type of problems. As the focus of this work is multi-class classification, I am introducing rectified linear unit (ReLU) activation, softmax activation, and cross entropy loss functions which are used in NN to solve multi-class classification problems.

2.1.4.1. ReLU

The function is relatively simple, $f(x) = \max(x, 0)$.

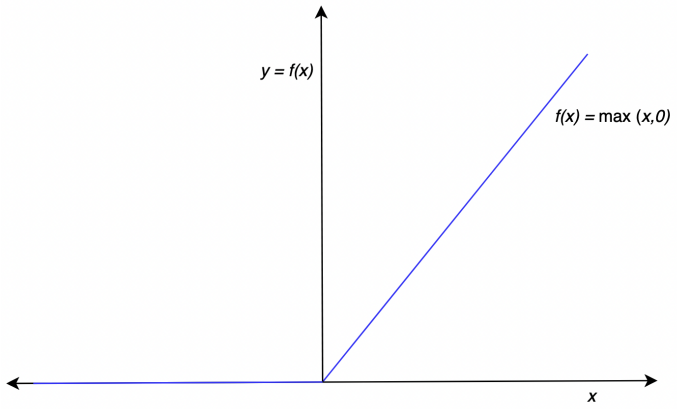

Figure 3: ReLU activation function

In other words, it is also represented as:

$$y = f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{5}$$

In short, it can also be written as $f(x) = (x > 0).x$ where $(x > 0)$ is the result bit of the comparison. Then, if we call the derivative of ReLU function as DReLU, it is represented as:

$$y = f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{6}$$

2.1.4.2. Softmax

Given an input vector x, softmax activation is a way to normalize the components of the vector. Each component of the vector, x_i, is normalized by applying:

$$g(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}} \tag{7}$$

During backpropagation while computing the partial derivative $\frac{\partial J}{\partial w}$, the derivative of softmax is required which is:

$$
\begin{aligned}
\frac{\partial g(x_i)}{\partial x_j} &= \frac{\sum_i e^{x_i} \times e^{x_i}\{i=j\} - e^{x_i} \times e^{x_j}}{\left(\sum_i e^{x_i}\right)^2} \\
&= \frac{\sum \times e^{x_i}\{i=j\} - e^{x_i} \times e^{x_j}}{\left(\sum\right)^2} \qquad \left(\text{writing } \sum_i e^{x_i} \text{ as } \sum\right) \\
&= \frac{e^{x_i}\left(\sum\{i=j\} - e^{x_j}\right)}{\left(\sum\right)^2} \\
&= \frac{e^{x_i}}{\sum} \times \frac{\left(\sum\{i=j\} - e^{x_j}\right)}{\sum} \\
&= g(x_i)[1\{i=j\} - g(x_j)]
\end{aligned}
$$

So, in conclusion, the softmax derivative can be written as:

$$
\frac{\partial g(x_i)}{\partial x_j} = \begin{cases} g(x_i)[1 - g(x_j)] & i = j \\ -g(x_i) \times g(x_j) & i \neq j \end{cases} \tag{8}
$$

2.1.4.3. Cross Entropy Loss

The cross entropy loss function is defined as:

$$
J(w) = -\sum_i^n y_i \log(\hat{y_i}) \tag{9}
$$

where n is number of data points, $(\hat{y_i})$ is the predicted output with its corresponding target label y_i for input data x_i.

2.1.5. Neural Networks

Linear regression is able to model the relationship in the data that is linear whereas logistic regression is able to solve binary classification problem but none of these two are able to model relationship in data that is non-linear. Hence, a more robust function approximation algorithms like neural network (NN) is used which is capable of modelling any sort of function: linear and non-linear. They are able to do so because they are a universal approximator (see [4] pg 192-194).
Neural network is a collection of neurons (logistic regressions) arranged vertically in different layers as shown in the Figure 4 where each neuron gets inputs $x \in \mathbb{R}^{d_l}$ and has weights $w \in \mathbb{R}^{d_l}$ to get the weighted sum $s = x.w$ and applies some activation function f on the weighted sum s to get the final output $\hat{y} = g(s)$ or the input for the next layer $z = f(s) = f(x.w)$. A neural network has an input layer, several hidden layers each having certain number of neurons, and an output layer. The number of layers and the number of neurons in each layer to keep in a neuron network is determined by heuristics depending on what combination gives you the better results. But the general rule is the more the better as more hidden layers are able to model complex relationship in the given data.

2.1.5.1. Notation

Considering a data point (x, y) among n data points, the notation of the components of an NN is defined as:

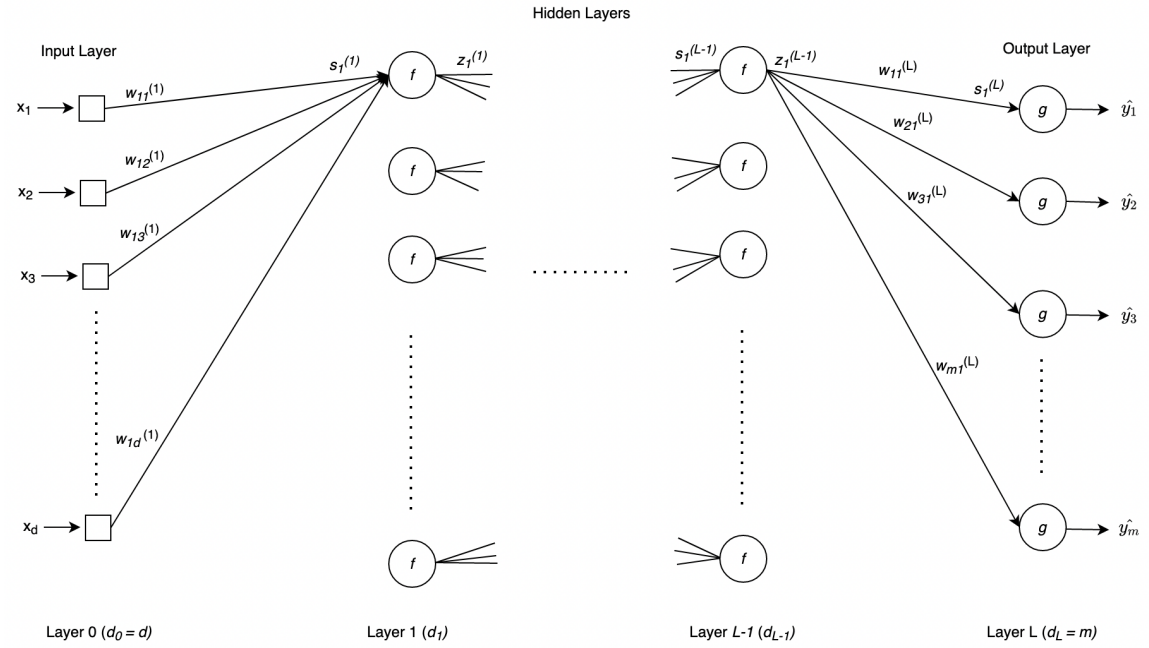

Figure 4: Neural Network Architecture

Layers: the layers are denoted by $1 \leq l \leq L$ where the input layer is not included. An NN with L layers means the total number of hidden layers and output layers is L with specifically $L-1$ hidden layers.

Neurons: the number of neurons at each layer is denoted by d_l and at input layer we have $d_0 = d$ features, and at the output layer we have $d_l = m$ classes. Then, $0 \leq i \leq d_l$ and l denote the i-th neuron of layer l.

Inputs: inputs at the input layer are denoted by $x_i \in \mathbb{R}$ where $0 \leq i \leq d$ is the i-th feature of input vector x. Similarly, inputs at the subsequent layers are denoted by s_i^l where i and l denoted the input to i-th neuron of layer l.

Weights: each neuron at layer l has a weight associated with each neuron layer $l-1$, making the weight vector at layer l, W_l, of dimension $l \times (l-1)$. The weight at layer l connecting the j-th neuron of layer l with i-th neuron of layer $(l-1)$ is denoted by w_{ji}^l.

Outputs: output at the output layer are denoted by $\hat{y_k} \in \mathbb{R}^m$ where $0 \leq k \leq m$ is the k-th class of the output vector $\hat{y}$ where its corresponding target label is denoted as y_k. Similarly, inputs at the previous layers are denoted by z_i^l where i and l denoted the input of i-th neuron of layer l.

2.1.5.2. Forward Propagation

During forward propagation, the input to the each layer l is the weighted sum (s_j^l)

$$s_j^l = \sum_{i=0}^{d_{l-1}} w_{ji} \times z_i^{l-1} \tag{10}$$

where z_i^{l-1} is the output of the previous layer $(l-1)$. For the first layer $l = 1$, the inputs are $z_i = x_i$. The output of the each layer l is computed by applying an activation function

f on the weighted sum (s_i^l). In other words,

$$z_i^l = f(s_i^l) \tag{11}$$

For the last layer $l = L$, a softmax function (g) is applied to get the final output $\hat{y}_k = z_k^L = g(s_k^L)$. So, if there are m output classes, then output of each neuron, $1 \leq k \leq m$, is represented as

$$\begin{aligned}
\hat{y}_k &= g(s_k^L) \\
&= g\left(\sum_{j=0}^{d_{L-1}} w_{kj} \times z_j^{L-1}\right) \\
&= g\left(\sum_{j=0}^{d_{L-1}} w_{kj} \times f\left(\ldots\ldots f\left(\sum_{h=0}^{d} w_{ih} \times x_h\right)\right)\right)
\end{aligned}$$

2.1.5.3. Backward Propagation

To update the weights at each layer l, partial derivative with respect to weights, $\frac{\partial J}{\partial W^l}$, is computed using the backpropagation algorithm. Especially for neural networks, the backpropagation rule is well summarized by the delta rule. To introduce delta rule, let's define the error at neuron j of layer l as $\delta_j^l = \frac{\partial J}{\partial s_i^l}$ which is the partial derivative of loss function (J) with respect to the weighted sum (s_j^l) at layer l. Then, the error (δ_j^l) can be expanded as:

$$\delta_j^l = \frac{\partial J}{\partial s_i^l} = \frac{\partial J}{\partial z_i^l} \times \frac{\partial z_i^l}{\partial s_i^l} = \frac{\partial J}{\partial z_i^l} \times f'(s_i^l) \tag{12}$$

Now, the delta rule gives the error (δ^l) at layer l in terms of error at the next layer ($l+1$) as

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot f'(s^l) \tag{13}$$

where $\odot$ is the element-wise product. Then, the partial derivative of J with respect to weights w_{ji}^l of neuron j at layer l in terms of delta is

$$\frac{\partial J}{\partial w_{ji}^l} = \frac{\partial J}{\partial s_j^l} \times \frac{\partial s_j^l}{\partial w_{ji}^l} = \delta_j^l \times z_i^{l-1} \tag{14}$$

2.1.5.4. Update Function

For a neural network, the update function at each layer can be found combining equation 13 and equation 14. I have worked out the update function for any layer for a neural network with logistic activation function in the hidden layer and softmax activation function in the output layer in my extended notes (see Notes in [10]).

For this work, I have considered a neural network with two layers (one hidden and one output). Let's call the neural network as K. K has 256 neurons in the hidden layer with ReLU as the activation function. I am training my NN model on MNIST[3] dataset which has 10 output classes, hence K has ten output neurons in the output layer with softmax as the activation. The loss function (J) used is Cross entropy function (9). The neural network K is similar to the neural network A considered in SecureNN ([11]).

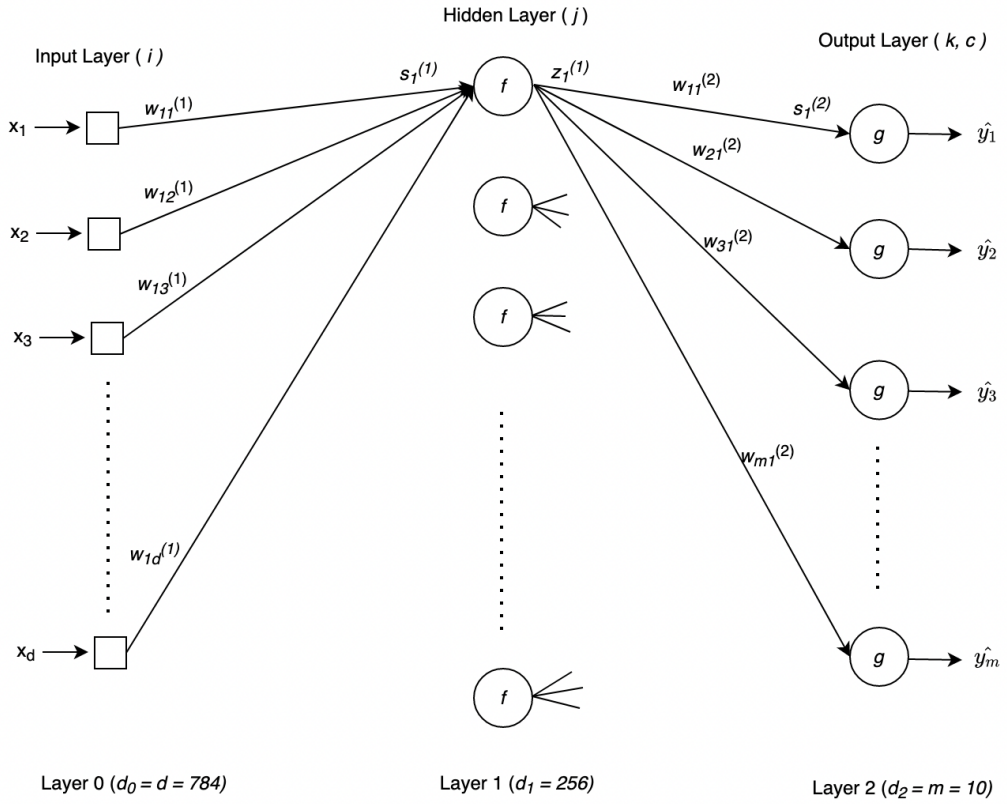

Figure 5: Neural Network K

Let's compute the update function for the neural network K. The neurons at the output layer are denoted by $1 \leq k \leq m = 10$ and $1 \leq c \leq m = 10$, at hidden layer denoted by $1 \leq j \leq d_1 = 256$, and the inputs denoted by $1 \leq i \leq 784$.

i) For the output layer(L):

$$
\begin{aligned}
\frac{\partial J}{\partial w_{kj}} &= \frac{\partial J}{\partial s_c} \times \frac{\partial s_c}{\partial w_{kj}} \\
&= \frac{\partial J}{\partial s_c} \times z_j && \text{(Taking derivative on eqn 10)} \\
&= \frac{\partial}{\partial s_c} \left(-\sum_{k=1}^{m} y_k \log \hat{y}_k \right) \times z_j && \text{(From eqn 9)} \\
&= \left(-\sum_{k=1}^{m} y_k \frac{\partial \log \hat{y}_k}{\partial s_c} \right) \times z_j \\
&= \left(-\sum_{k=1}^{m} y_k \times \frac{1}{\hat{y}_k} \times \frac{\partial \hat{y}_k}{\partial s_c} \right) \times z_j \\
&= \left(-\sum_{k=1}^{m} y_k \times \frac{1}{\hat{y}_k} \times \hat{y}_k (1\{k = c\} - \hat{y}_c) \right) \times z_j && \text{(From eqn 7)} \\
&= \left(-\sum_{k=1}^{m} y_k \times (1\{k = c\} - \hat{y}_c) \right) \times z_j
\end{aligned}
$$

$$
\begin{aligned}
&= \left(\left(\sum_{k \neq c}^{m} y_k \hat{y_c} \right) - y_c(1 - \hat{y_c}) \right) \times z_j \\
&= \left(\left(\sum_{k \neq c}^{m} y_k \hat{y_c} \right) - y_c + y_c \hat{y_c}) \right) \times z_j \\
&= \left(\left(\sum_{k=1}^{m} y_k \hat{y_c} \right) - y_c \right) \times z_j \\
&= \left(\left(\hat{y_c} \sum_{k=1}^{m} y_k \right) - y_c \right) \times z_j \\
&= (\hat{y_c} - y_c) \times z_j \qquad \text{(sum of probabilities is } 1 \sum_{k=1}^{m} y_k = 1)
\end{aligned}
$$

Then, from the derivation, the delta error(δ_k^L) for neurons in the output layer (L) is

$$\delta_k^L = \frac{\partial J}{\partial s_k} = (\hat{y_k} - y_k) \tag{15}$$

Then, the update function for weights(w_{kj}) in the output layer is:

$$w_{kj}(new) := w_{kj}(old) - \alpha[z_j(\hat{y_k} - y_k)] \tag{16}$$

ii) For the hidden layer($L-1$):

$$
\begin{aligned}
\frac{\partial J}{\partial w_{ji}} &= \sum_{k=1}^{m} \frac{\partial J}{\partial s_k} \times \frac{\partial s_k}{\partial z_j} \times \frac{\partial z_j}{\partial s_j} \times \frac{\partial s_j}{\partial w_{ji}} \\
&= \sum_{k=1}^{m} \delta_k^L \times w_{kj} \times f'(s_j) \times x_i \qquad \text{(From eqn10 and 11} \\
&= \left(\sum_{k=1}^{m} (\hat{y_k} - y_k) \times w_{kj} \right) \times f'(s_j) \times x_i \qquad \text{(From eqn15)}
\end{aligned}
$$

Then, the update function for weights(w_{ji}) in the hidden layer is:

$$w_{ji}(new) := w_{ji}(old) - \alpha \frac{\partial J}{\partial w_{ji}} \tag{17}$$

where $\frac{\partial J}{\partial w_{ji}}$ is computed above.

2.1.5.5. NN building blocks

In conclusion, for any neural network that solves a multi-class classification problem, the main computation of the NN apart from addition/subtraction involves:

i) Multiplication

ii) Computation of non-linear activation function. In our case, computation of ReLU and Softmax in the NN K (5) and NN A from SecureNN [11].

iii) Division

For privacy preserving neural network (PPNN), these three operation should be computed securely in the SOC setting.

2.1.5.6. Implementation:

I have implemented the neural network K described above from scratch in vectorized way and trained it on the MNIST [3] dataset using mini-batch method. The implementation can be found here [10].

Dataset Description:

Dataset	**Training Samples**	**Testing Samples**	**Features (d)**	**Range**
MNIST [3]	60000	10000	784	$x \in [0, 255], y \in [0, 9]$

Table 1: Implemented Dataset Details

Training and Testing Results: the total training time was about 30-40 minutes and $\alpha = 0.01$

Epoch	**Training Loss**	**Training Accuracy**
1	6.29234	67.42 %
2	2.43125	82.245 %
3	1.83954	85.1833 %
4	1.5275	86.77 %
5	1.32303	87.7933 %
6	1.17372	88.5933 %
7	1.05836	89.255 %
8	0.965678	89.7567 %
9	0.888937	90.1617 %
10	0.824245	90.5917 %
11	0.768885	90.9433 %
12	0.720802	91.255 %
13	0.678561	91.5667 %
14	0.640987	91.8117 %
15	0.607362	92.0233 %

Table 2: Implementation Results

Testing Loss: 0.705534 and **Testing Accuracy:** 91.23 %

2.2. Secure Computation

2.2.1. Secret Sharing

Secret sharing refers to the method of splitting a secret into multiple shares, and distributing the shares among the multiple parties, so that only when the parties come together the secret can be reconstructed from the shares. The holder of the secret which is referred as dealer creates n shares of a secret and defines a threshold t for the number of shares required to reconstruct the shares. The dealer then distribute the shares to the n parties. Such a system is called (n, t) scheme.
For PPML frameworks, the dealer is obviously the data owners, and the number of parties n depends on the number of servers in the SOC setting. For this work, SecureML[7] and [11] considers secret sharing among two servers, specifically 2-out-of-2 additive secret sharing over three rings $\mathbb{Z}_L$, $\mathbb{Z}_{L-1}$ and $\mathbb{Z}_p$ where $L = 2^{64}$, $l = 64$ and $p = 67$. Note that Z_{L-1} is an ring of odd order and $\mathbb{Z}_p$ is a field. $\langle x \rangle_i^t$ is used to denote the i-th share of x over $\mathbb{Z}_t$. The algorithm $\texttt{Share}^t(x)$ generates two shares of x over $\mathbb{Z}_t$ and $\texttt{Rec}^t(x)$ reconstructs the two shares of x over $\mathbb{Z}_t$

2.2.1.1. 2-out-of-2 additive secret sharing in $\mathbb{Z}_L$

Two shares of x over $\mathbb{Z}_L$ are denoted as $(\langle x \rangle_0^L, \langle x \rangle_1^L)$

- Input: $x \in \mathbb{Z}_L$
- $\texttt{Share}^L(x)$:
  1. Pick a common random $r \in \mathbb{Z}_L$
  2. Set first share as $\langle x \rangle_0^L = \mathrm{r}$
  3. Set second share as $\langle x \rangle_1^L = (x - r) \bmod L$
- Output: $(\langle x \rangle_0^L, \langle x \rangle_1^L)$
- $\texttt{Rec}^L(\langle x \rangle_0^L, \langle x \rangle_1^L)$: $x = \langle x \rangle_0^L + \langle x \rangle_1^L \bmod L$

2.2.1.2. 2-out-of-2 additive secret sharing in $\mathbb{Z}_{L-1}$

Two shares of x over $\mathbb{Z}_{L-1}$ are denoted as $(\langle x \rangle_0^{L-1}, \langle x \rangle_1^{L-1})$

- Input: $x \in \mathbb{Z}_{L-1}$
- $\texttt{Share}^{L-1}(x)$:
  1. Pick a common random $r \in \mathbb{Z}_{L-1}$
  2. Set first share as $\langle x \rangle_0^{L-1} = \mathrm{r}$
  3. Set second share as $\langle x \rangle_1^{L-1} = (x - r) \bmod (L - 1)$
- Output: $(\langle x \rangle_0^{L-1}, \langle x \rangle_1^{L-1})$
- $\texttt{Rec}^{L-1}(\langle x \rangle_0^L, \langle x \rangle_1^{L-1})$: $x = \langle x \rangle_0^{L-1} + \langle x \rangle_1^{L-1} \bmod (L - 1)$

2.2.1.3. 2-out-of-2 additive secret sharing in $\mathbb{Z}_p$

Two shares of x over $\mathbb{Z}_p$ are denoted as $(\{\langle x[i]\rangle_0^p\}_{i\in[l]}, \{\langle x[i]\rangle_1^p\}_{i\in[l]})$ where $l = 64$

- Input: $x \in \mathbb{Z}_L$ is a 64-bit integer
- $\texttt{Share}^p(x)$:
  1: **for** $i = \{0, \ldots, l\}$ **do**
  2: Pick a common random $r \in \mathbb{Z}_p$
  3: Set $\langle x[i]\rangle_0^p = r$
  4: Set $\langle x[i]\rangle_1^p = (x[i] - r) \bmod p$
  5: **end for**
- Output: $(\{\langle x[i]\rangle_0^p\}_{i\in[l]}, \{\langle x[i]\rangle_1^p\}_{i\in[l]})$
- $\texttt{Rec}^p(\{\langle x[i]\rangle_0^p\}_{i\in[l]}, \{\langle x[i]\rangle_1^p\}_{i\in[l]})$:
  1: **for** $i = \{0, \ldots, l\}$ **do**
  2: $x[i] = (\langle x[i]\rangle_0^p + \langle x[i]\rangle_1^p) \bmod p$
  3: **end for**

2.2.2. Beaver's Multiplication Triples

A beaver's multiplication triples[1] is a tuple of secret-shared values $([x], [y], [z])$ such that $z = x \times y$, and x and y are uniformly chosen at random. Generally, the whole of $x, y, andz$ is not known to any one party in MPC: the triples are precomputed and shared to the parties. But in PPML frameworks, the beaver's triples are generated by the servers using common randomness in their offline phase to be used later in the online phase.

2.2.3. Arithmetic Operations on Shared Decimal Numbers

Almost all protocol in cryptography and especially in secure multiparty computation (MPC) work with integer rings whereas the real world data are real numbers and hence, all ML algorithms carry out their computation on decimal numbers. Therefore, first of all, a mapping that maps the input data in decimal numbers into integer rings ($\mathbb{Z}_\mathbb{L}$). Recent PPML secure frameworks such as SecureML[7], BLAZE [9], and SecureNN [11] use a mapping that maps the fixed point decimal number to a $\mathbb{Z}_{2^{64}}$ integer ring ($L = 2^{64}$). The integer ring $\mathbb{Z}_{2^{64}}$ is chosen so that while implementing we could represent the integers in the native C++ datatype *uint_64* which stores 64-bit unsigned integers. Let's talk first about fixed point decimal numbers representation.

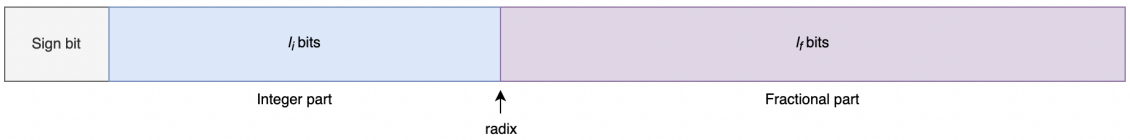

Figure 6: Fixed Point Representation

In fixed point representation, the decimal numbers are stored in *2's complement* format. The most significant bit (MSB) is reserved for indicating the sign of the number, '0' indicating a positive number and '1' indicating a negative number. Then, l_i bits are reserved for

representing the integer part and l_f bits for representing the fractional part. Additionally, the separation between the integer and fractional part is called the radix point as shown in the Figure 6.

2.2.3.1. Mapping

Consider a fixed-point decimal number x with at most l_f fractional bit, then the mapping of x to $\mathbb{Z}_{2^{64}}$ is done by multiplying it with 2^{l_f}. So, if x' is the mapping of x in $\mathbb{Z}_{2^{64}}$, then $x' = 2^{l_f}x$. For negative numbers it works slightly differently, if y' is a mapping of a negative fixed point decimal number y, then $y' = 2^{64} - |2^{l_f}y|$.

Algorithm 1 Mapping $\prod_{Map}(x)$

Input: $x \in \mathbb{R}$
Output: $x' \in \mathbb{Z}_{2^{64}}$ is the mapping of x

1: **if** $x \geq 0$ **then**
2: $\quad x' \leftarrow (2^{l_f} \times x)$
3: **else**
4: $\quad x' = 2^{64} - |2^{l_f} \times x|$
5: **end if**

The 2^{l_f} is the precision of decimal numbers that is preserved during the mapping and can be taken lower than 2^{l_f} depending on the requirement of accuracy in the computation. To that end, 2^{l_f} is known as the scaling factor by which you scale the decimal numbers before mapping to $\mathbb{Z}_{2^{64}}$.

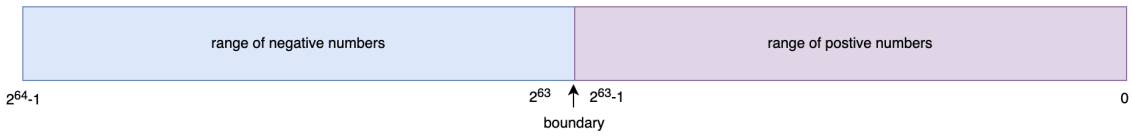

Figure 7: Integer ring $\mathbb{Z}_{2^{64}}$

After mapping, the range of positive number that is supported in the ring is from 0 to $2^{63} - 1$ and the negative number is supported from 2^{63} to $2^{64} - 1$ as shown in the Figure 7 . All operations in the ring occur over mod $\mathbb{Z}_{2^{64}}$, so any number that is higher than $2^64 - 1$ will simply wrap around the integer ring.

2.2.3.2. Truncation

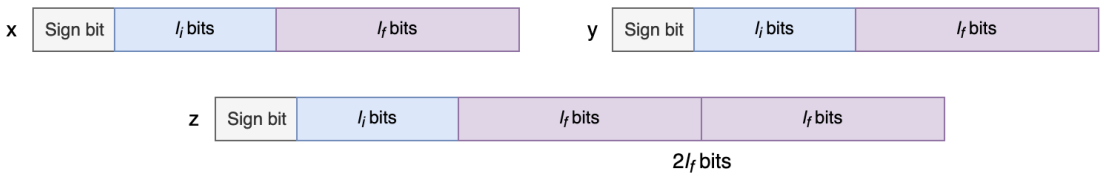

Figure 8: Multiplication of two fixed decimal points doubles the fractional part

The addition of two fixed point decimal numbers is straightforward, first map them to the ring and add them. But the multiplication of two fixed point decimals require additional operations. Consider the fixed-point multiplication of two decimal numbers x and y with at most l_f fractional bits. First, map them to integer ring, $x' = 2^{l_f}x$ and $y' = 2^{l_f}y$. Then, the product $z = x'y'$ has at most $2l_f$ bits representing the fractional part of the product, and to keep at most l_f bits for z, the last l_f bits of z is simply truncated as shown in the Figure 8.

Algorithm 2 Truncation $\prod_{Trun}(x)$

Input: $z \in \mathbb{Z}_{2^{64}}$ with at most $2l_f$ fractional bits
Output: $z \in \mathbb{Z}_{2^{64}}$ with at most l_f fractional bits

1: $z \leftarrow \frac{z}{2^{l_f}}$

The above truncation technique also works for the 2-out-of-2 secret shares. The two parties simply needs to truncate their shares to truncate a secret preserving the correctness with a error of at most 1 bit with high probability. This is shown in theorem 1 of SecureML paper[7].

3. SecureNN

3.1. Technical Overview

As discussed in section 2.1.5.4, to build a privacy preserving neural network (PPNN) with ReLU and Softmax, we need to compute three building block/operations privately/securely over input data in the integer ring $\mathbb{Z}_L$, namely multiplication, computation of non-linear activation functions, and division.

For secure matrix multiplication, beaver multiplication triples is used to carry out multiplication on two secret shares in two party setting in SecureML[7] paper. The same multiplication technique is also used in the SecureNN [11] paper. A brief description of the secure matrix multiplication from SecureNN which is described in the section 3.2.1

For computation of `ReLU(`a`)`, SecureNN details `Compute MSB` functionality, the details of which are mentioned in the section 3.2.4. But the idea is that to compute `ReLU(`a`)`, it suffices to compute its derivative as suggested by the equation 5, the derivative of `ReLU` tells computes $(a > 0)$ and tell us whether the number is positive or negative. In essence, `ReLU(`a`)` $= (a > 0) \cdot a =$ `ReLU'(`a`)` $\cdot\, a$. Now, it must be noted that finding `ReLU'(`a`)` is closely related to finding the `MSB` of a in the integer ring $\mathbb{Z}_L$. That is, `ReLU'(`a`)` $= 1$ iff `MSB(`a`)` $= 0$. Hence, it suffice to calculate `MSB(`a`)` to compute `ReLU(`a`)`.

The idea to compute the `MSB(`a`)` in the SecureNN paper is to convert the `MSB(`a`)` to `LSB(`$2a$`)` computation over the odd ring $\mathbb{Z}_{L-1}$ where `LSB` is the least significant bit. The key idea here is that in a group of order n, we have `MSB(`a`)` $= 1$ iff $a > \frac{n}{2}$ iff $n > 2a - n > 0$; if n is odd, then so is $2a - n$ and it follows that `MSB(`a`)` $= 1$ iff `LSB(`$2a$`)`.

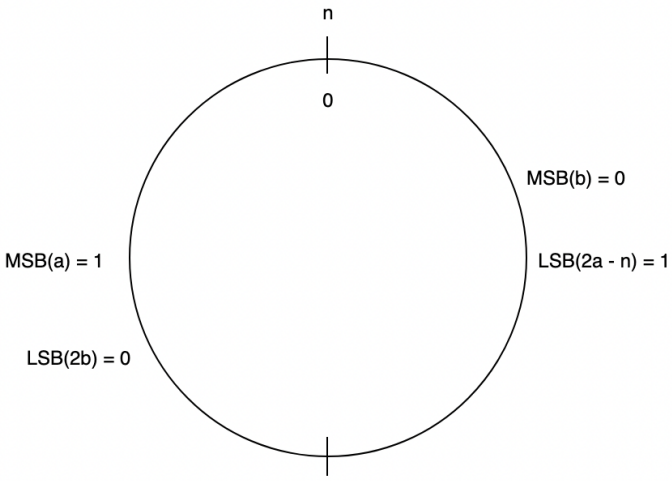

Figure 9: Odd integer ring of order n

So to find the `MSB(`a`)`, shares of a over $\mathbb{Z}_L$ must be converted to an odd ring which is $\mathbb{Z}_{L-1}$. For now, let's assume we know how to convert the shares of a over $\mathbb{Z}_{2^{64}}$ to over $\mathbb{Z}_{2^{64}-1}$. SecureNN provides `Share Convert` functionality to convert the shares: the details of the functionality is mentioned in the section 3.2.3. Then, at the start of the protocol, P_0 and P_1 hold shares of a over $\mathbb{Z}_{2^{64}-1}$, and they locally compute the shares of $y = 2a$ in order to find out `LSB(y)` $= y[0]$. Now, to find out $y[0]$, P_0 and P_1 can not simply exchange the secret shares of y among themselves as it would reveal the original value of x. Hence, they both need to mask the shares before sharing. Let a common random number x be the mask. P_2

generates shares of x and gives them to P_0 and P_1. P_0 and P_1 then compute the shares of $r = y + x$, exchange the share and reconstruct r. Here, the first observation is that for three u, v and w such that $u = v + w, u[0] = v[0] \oplus w[0]$ if the addition doesn't wrap around the ring but if it does, then $u[0] = v[0] \oplus w[0] + 1$ because we are operating over and odd ring and the parity of the sum (u) changes when it wraps around the ring.

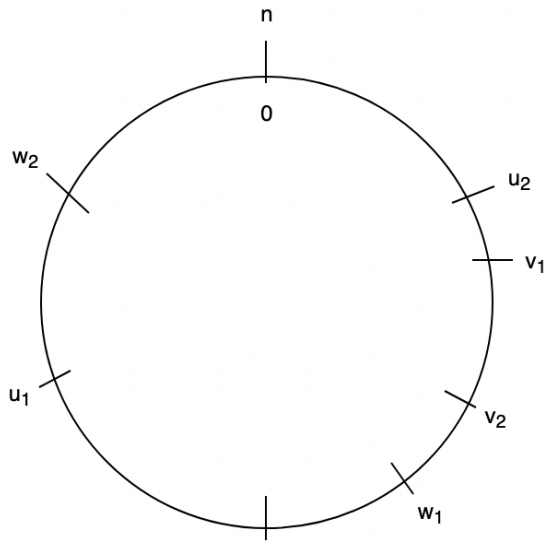

Figure 10: Addition of three number in the odd ring

For instance, if the u is *even*, then *even* mod *odd* (n) will be *odd* and if u is *odd*, then *odd* mod *odd* (n) will be *even*. So, to negate the parity change we add '1' to the sum. Moreover, if γ is the wrap bit which tells whether r wraps around the odd ring $\mathbb{Z}_{L-1}$ or not, then P_0 and P_1 need to compute shares of `LSB(y)` $= y[0]$ as $y[0] = r[0] \oplus x[0] \oplus \gamma$.

Now, when P_2 generates secret shares of x over $\mathbb{Z}_{2^{64}}$, it also generates the secret shares of $x[0]$ over $\mathbb{Z}_{2^{64}-1}$ and over $\mathbb{Z}_p$, that is $\langle x[0] \rangle_j^L$ and $\{\langle x[i] \rangle_j^p\}_{i \in [l]}$ for $j \in \{0, 1\}$, for P_0 and P_1. Then, P_0 and P_1 can construct secret shares of $r[0]$. Now, in order to compute secret shares of $y[0]$, all we have to figure out is a way to compute secret shares of γ, in other words, whether the addition $r = y + x$ wraps around the ring or not. For this, the second observation is that the addition wraps around iff the sum (r) is less than one of the individual operand, x or y - that is, it wraps around iff $x > r$. Hence, we just need to compute the shares of $(x > r)$ between P_0 and P_1 to compute shares of γ, and we are done. To that end, $\gamma = (x > r)$ bit and `LSB(y)` $= y[0]$ as $y[0] = r[0] \oplus x[0] \oplus (x > r)$.

To compute $x > r$, SecureNN presents `Private Compare` functionality, the details of which is mentioned in the section 3.2.2. `Private Compare` is a three-party functionality where P_0 and P_1 have l-bit secret shares of x over field $\mathbb{Z}_p$, a common number r (which was reconstructed above), and a random mask bit β. Then, it computes the bit $(x > r)$ and masks the output with β and gives the result $(\beta \oplus (x > r))$ to P_2. SecureNN builds the `Private Compare` functionality on top of technique used in [2, 8]. With the `Compute MSB` and `Private Compare`, we can now compute `ReLU'(`a`)` and consequently `ReLU'(`a`)`.

To compute `Softmax(`x_i`)` where x_i's are components of a vector x, SecureML approximates the natural exponentiation function e^{x_i} with `ReLU(`x_i`)` as shown in the figure 11.

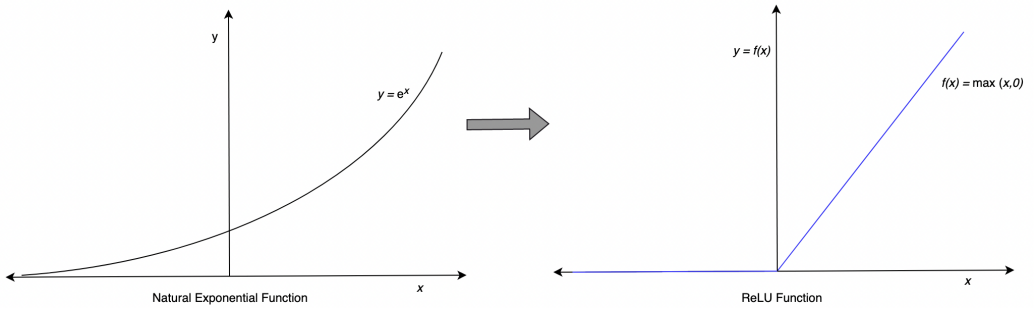

Figure 11: Exponential to ReLU approximation

Then, the approximated softmax becomes:

$$\texttt{ASM}(x_i) = \frac{\texttt{ReLU}(x_i)}{\sum_i \texttt{ReLU}(x_i)} \tag{18}$$

SecureNN too uses the same approximation.Then, only thing left to figure out is to carry out operation in the $\mathbb{Z}_L$ ring. SecureNN provides `Division` functionality which does long division (normal division) where the quotient is computed bit-by-by sequentially from the `MSB` as any number can be written as a sum of powers of 2. The details of the functionality is mentioned is section 3.3.3. The figure 12 shows how the dependence of supporting and main protocols between each other.

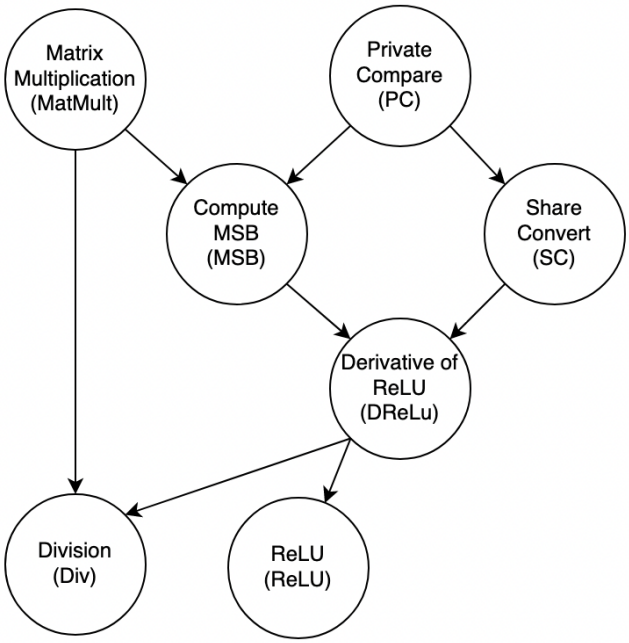

Figure 12: Dependency between protocols

3.2. Supporting Protocols

3.2.1. Matrix Multiplication

Matrix Multiplication ($\mathcal{F}_{\text{MatMult}}$) provides the functionality of securely multiplying two numbers in a ring between parties P_0 and P_1. P_0 and P_1 start with shares of $X \in \mathbb{Z}_L^{m \times n}$ and $Y \in \mathbb{Z}_L^{n \times v}$ respectively and get shares of $Z = X \cdot Y$ at the end.

Algorithm 3 Matrix Multiplication $\prod_{\text{MatMult}}(\{P_0, P_1\}, P_2)$

H

Input: P_0 and P_1 hold $(\langle X \rangle_0^L, \langle Y \rangle_0^L)$ and $(\langle X \rangle_1^L, \langle Y \rangle_1^L)$ respectively.
Output: P_0 gets $(\langle X.Y \rangle_0^L)$ and P_1 gets $(\langle X.Y \rangle_1^L)$
Common Randomness: P_0 and P_1 hold shares of zero matrices over $\mathbb{Z}_L^{m \times v}$ respectively; that is, P_0 holds $\langle 0^{m \times v} \rangle_0^L = U_0$ and P_1 holds $\langle 0^{m \times v} \rangle_1^L = U_1$

1: P_2 picks random matrices $A \xleftarrow{\$} \mathbb{Z}_L^{m \times n}$ and $B \xleftarrow{\$} \mathbb{Z}_L^{n \times v}$ and generates for $j \in \{0, 1\}, \langle A \rangle_j^L, \langle B \rangle_j^L, \langle C \rangle_j^L$ and sends to P_j where $C = A \cdot B$.
2: For $j \in \{0, 1\}, P_j$ computes $\langle E \rangle_j^L = \langle X \rangle_j^L - \langle A \rangle_j^L$ and $\langle F \rangle_j^L = \langle Y \rangle_j^L - \langle B \rangle_j^L$.
3: P_0 and P_1 reconstruct E and F by exchanging shares.
4: For $j \in \{0, 1\}, P_j$ outputs $-jE \cdot F + \langle X \rangle_j^L \cdot F + E \cdot \langle Y \rangle_j^L + \langle C \rangle_j^L + U_j$.

Explanation: The protocol carries out standard cryptographic technique for multiplication using Beaver multiplication triple [1] which are generated by P_2 and given to parties P_0 and P_1. In the step 4 of the protocol, P_0 has $\langle Z \rangle_0^L = \langle X \rangle_0^L \cdot F + E \cdot \langle Y \rangle_0^L + \langle C \rangle_0^L + U_0$ and P_1 has $\langle Z \rangle_1^L = -E \cdot F + \langle X \rangle_1^L \cdot F + E \cdot \langle Y \rangle_1^L + \langle C \rangle_1^L + U_1$. Then, if we add both the outputs we get:

$$
\begin{aligned}
\langle Z \rangle_0^L + \langle Z \rangle_1^L &= \langle X \rangle_0^L \cdot F + E \cdot \langle Y \rangle_0^L + \langle C \rangle_0^L + U_0 - E \cdot F + \langle X \rangle_1^L \cdot F + E \cdot \langle Y \rangle_1^L \\
&\quad + \langle C \rangle_1^L + U_1 \\
&= (\langle X \rangle_0^L + \langle X \rangle_1^L) \cdot F + E \cdot (\langle Y \rangle_0^L + \langle Y \rangle_0^L) + C + U - E \cdot F \\
&= X \cdot F + E \cdot Y + C - E \cdot F \\
&= X \cdot F + (X - A) \cdot Y + A \cdot B - E \cdot F && \text{(From step 2)} \\
&= X \cdot F + X \cdot Y - A \cdot Y + A \cdot B - E \cdot F \\
&= X \cdot F + X \cdot Y - A \cdot (Y - B) - E \cdot F \\
&= X \cdot F + X \cdot Y - A \cdot F - E \cdot F && \text{(From step 2)} \\
&= (X - A) \cdot F + X \cdot Y - E \cdot F \\
&= E \cdot F + X \cdot Y - E \cdot F && \text{(From step 2)} \\
&= X \cdot Y \\
&= Z
\end{aligned}
$$

This shows how the standard cryptographic technique for multiplication is able to carry out multiplication between shares of two numbers securely.

3.2.2. Private Compare

Private Compare ($\mathcal{F}_{\text{PC}}$) provides the functionality of comparing two numbers (say x and r) in the integer ring. The protocol is a 3-party protocol where P_0 and P_1 start with

shares of bits of l-bit integer x over $\mathbb{Z}_p$ - that is, $\{\langle x[i]\rangle_0^p\}_{i\in[l]}, \langle x[i]\rangle_1^p\}_{i\in[l]}$ respectively. They also hold an l-bit integer r against which x is compared, $(x > r)$ and a common random bit β. The result of the comparison is masked and learnt byP_2 which is $\beta' = \beta(x > r)$. It is very crucial to note that Private Compare functionality compares two numbers in the integer ring independent of what their mapping to real numbers is.

Algorithm 4 Private Compare $\prod_{\text{PC}}(\{P_0, P_1\}, P_2)$

Input: P_0 and P_1 hold $\{\langle x[i]\rangle_0^p\}_{i\in[l]}, \langle x[i]\rangle_1^p\}_{i\in[l]}$ respectively, a common input r (an l-bit integer) and a common random bit β.
Output: P_2 gets a bit $\beta \oplus (x > r)$.
Common Randomness: P_0 and P_1 hold l common random values $s_i \in \mathbb{Z}_p^*$ for all $i \in [l]$ and a random permutation π for l elements. P_0 and P_1 additionally hold l common random values $u_i \in \mathbb{Z}_p^*$.

1: Let $t = r + 1 \bmod 2^l$.
2: For each $j \in \{0, 1\}, P_j$ executes Steps 3-14:
3: **for** $i = \{l, l-1, \ldots, 1\}$ **do**
4: **if** $\beta = 0$ **then**
5: $\langle w_i\rangle_j^p = \langle x[i]\rangle_j^p + jr[i] - 2r[i]\langle x[i]\rangle_j^p$
6: $\langle c_i\rangle_j^p = jr[i] - \langle x[i]\rangle_j^p + j + \sum_{k=i+1}^{l} \langle w_k\rangle_j^p$
7: **else if** $\beta = 1$ `AND` $r \neq 2^l - 1$ **then**
8: $\langle w_i\rangle_j^p = \langle x[i]\rangle_j^p + jt[i] - 2t[i]\langle x[i]\rangle_j^p$
9: $\langle c_i\rangle_j^p = -jt[i] + \langle x[i]\rangle_j^p + j + \sum_{k=i+1}^{l} \langle w_k\rangle_j^p$
10: **else**
11: If $i \neq 1, \langle c_i\rangle_j^p = (1-j)(u_i + 1) - ju_i$, else $\langle c_i\rangle_j^p = (-1)^j \cdot u_i$
12: **end if**
13: **end for**
14: Send $\{\langle d_i\rangle_j^p\} = \pi\left(\left\{s_i\langle c_i\rangle_j^p\right\}\right)$ to P_2
15: For all $i \in [l], P_2$ computes $d_i = \texttt{Rec}^p(\langle d_i\rangle_0^p, \langle d_i\rangle_1^p)$ and sets $\beta' = 1$ iff $\exists\, i \in [l]$ such that $d_i = 0$
16: P_2 outputs β'

Explanation: SecureNN build Private Compare functionality on the top of techniques in [2, 8]. Depending on the common random mask bit β, the comparison $(x > r)$ changes. When $\beta = 0$, steps 4-6 straightforward compute $(x > r)$ but when $\beta = 1$, steps 7-9 computes $1 \oplus (x > r) \equiv (x \leq r) \equiv x < (r+1)$ over integers. For corner case when $r = 2^{64} - 1$, $(x \leq)$ is always true and the protocol constructs shares of c over Z_p such that output of comparison $(r + 1) > x$ is true - that is, $\beta' = 1$. So, when $\beta = 0$, β' is the output of comparison of $(x > r)$, and when $\beta = 1$, β' is the output of comparison of $(r + 1 > x)$. In both cases, $(x > r) = \beta' \oplus \beta$.

Let's try to how the comparison works without the mask bit β and secret shares of x over Z_p. It is obvious from the steps 5-6 and 8-9 that the comparison between x and r happens at the bit level; hence, the protocol expects boolean shares of bits of x over Z_p as inputs. Also, the protocol computes `XOR` of two bits (say $a \oplus b$) as $a \oplus b = a + b - 2ab \pmod 2$ in the protocol. Let's take two numbers $x = 141$ and $r = 135$ in the integer ring. Then, their binary representation is $x = 10001101$ and $r = 10000110$.

Clearly, $(x > r)$. Now, let's see how the protocol arrives at this result. Step 5 is computing $w = x \oplus r$ which is:

$$\begin{array}{r} 10001101 \\ \oplus 10000110 \\ \hline w = 00001011 \end{array}$$

And, $H = sum(w[i]) = 3$. Then, step 6 computes, for $i \in l, c[i] = r[i] - x[i] + 1 + \sum_{k=i+1}^{l} w_k$ and step 15 checks for a 0 in c. If $\exists$ a 0 in c, then $(x > r)$ is true. In this case $l = 8$. So,

$$\begin{aligned} c[8] &= (1-1) + 1 + 0 = 1 \\ c[7] &= (0-0) + 1 + 0 = 1 \\ c[6] &= (0-0) + 1 + 0 = 1 \\ c[5] &= (0-0) + 1 + 0 = 1 \\ c[4] &= (0-1) + 1 + 0 = 0 \quad \text{(point where they first differ)} \\ c[3] &= (1-1) + 1 + 1 = 2 \\ c[2] &= (1-0) + 1 + 1 = 3 \\ c[1] &= (1-0) + 1 + 2 = 3 \end{aligned}$$

There exists a 0 in c, so $x > r$.

Now, let's take an example with $x < r$, $x = 49 = 110001$ and $r = 53 = 110101$. Then, $w = x \oplus r$ is:

$$\begin{array}{r} 110001 \\ \oplus 110101 \\ \hline w = 000100 \end{array}$$

Again, $H = sum(w[i]) = 1$ and $l = 6$. So,

$$\begin{aligned} c[6] &= (1-1) + 1 + 0 = 1 \\ c[5] &= (1-1) + 1 + 0 = 1 \\ c[4] &= (0-0) + 1 + 0 = 1 \\ c[3] &= (1-0) + 1 + 0 = 2 \quad \text{(point where they first differ)} \\ c[2] &= (0-0) + 1 + 1 = 2 \\ c[1] &= (1-1) + 1 + 1 = 2 \end{aligned}$$

There does not exist a 0 in c, so $x \not> r$.

The thing to note in the computation of $c[i]$ is that when $x > r$ you get $r[i] - x[i] = -1$ at the place where they first differ. Let k denote the place(bit) where x and r first differ. Then, to obtained -1, we add 1 to it to get $-1 + 1 = 0$. Now, to preserve this '0' in the final c and distinguish it with other 0's in the computation, we add Hamming(H) distance till the $(k-1)^{th}$ step, which is also 0, to it. This gives us a 0 in our final c at the k^{th} bit. For the subsequent bits after k^{th} step where $(r[i] - x[i] + 1) = 0$, H isn't 0 anymore which ensures that we only get one 0 in our final c at the place where the numbers first differ, provided that $(x > r)$. This is the core idea of Private Compare functionality. Not to forget, the protocol compares the boolean shares of x over Z_p with bits of r.

3.2.3. Share Convert

Share Convert ($\mathcal{F}_{\text{SC}}$) provides the functionality to convert the shares of a number a from over $\mathbb{Z}_L$ to over $\mathbb{Z}_{L-1}$ where $L = 2^{64}$. It is a 3-part protocol where P_0 and P_1 start

with shares of a over L, $(\langle a\rangle_0^L, \langle a\rangle_1^L)$, such that $a \neq L-1$ and end with shares of a over $L-1$, $(\langle a\rangle_0^{L-1}, \langle a\rangle_1^{L-1})$, with the help of P_2. The protocol makes one call to $\mathcal{F}_{\text{PC}}$.

Algorithm 5 Share Convert $\prod_{\text{SC}}(\{P_0, P_1\}, P_2)$

Input: P_0 and P_1 hold $\langle a\rangle_0^L$ and $\langle a\rangle_1^L$ respectively such that $\texttt{Rec}^L(\langle a\rangle_0^L, \langle a\rangle_1^L) \neq L-1$.
Output: P_0 and P_1 get $\langle a\rangle_0^{L-1}$ and $\langle a\rangle_0^{L-1}$ respectively.
Common Randomness: P_0, P_1 hold a random bit η'', a random bit $r \in \mathbb{Z}_L$, shares $\langle r\rangle_0^L, \langle r\rangle_1^L, \alpha = \texttt{wrap}(\langle r\rangle_0^L, \langle r\rangle_1^L, L)$ and shares of 0 over $L-1$, denoted by u_0 and u_1.

1: For each $j \in \{0,1\}, P_j$ executes Steps 2-3.
2: $\langle \tilde{a}\rangle = \langle a\rangle_j^L + \langle r\rangle_j^L$ and $\beta = \texttt{wrap}(\langle r\rangle_0^L, \langle r\rangle_1^L, L)$
3: Send $\langle \tilde{a}\rangle_j^L$ to P_2.
4: P_2 computes $x = \texttt{Rec}^L(\langle \tilde{a}\rangle_0^L, \langle \tilde{a}\rangle_1^L)$ and $\delta = \texttt{wrap}(\langle \tilde{a}\rangle_0^L, \langle \tilde{a}\rangle_1^L, L)$
5: P_2 generates shares $\{\langle x[i]\rangle_j^P\}_{i\in[l]}$ and δ_j^{L-1} for $j \in \{0,1\}$ and sends to P_j.
6: P_0, P_1, P_2 invoke $\prod_{\text{PC}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having inputs $(\{\langle x[i]\rangle_j^P\}_{i\in[l]}, r-1, \eta'')$ and P_2 learns η'
7: For $j \in \{0,1\}, P_2$ generates $\langle \eta'\rangle_j^{L-1}$ and sends to P_j.
8: For each $j \in \{0,1\}, P_j$ executes Steps 9-11.
9: $\langle \eta\rangle_j^{L-1} = \langle \eta'\rangle_j^{L-1} + (1-j)\eta'' - 2\eta''\langle \eta'\rangle_j^{L-1}$
10: $\langle \theta\rangle_j^{L-1} = \beta_j + (1-j)\cdot(-\alpha - 1) + \langle \delta\rangle_j^{L-1} + \langle \eta\rangle_j^{L-1}$
11: Output $\langle a\rangle_j^{L-1} = \langle a\rangle_j^L - \langle \theta\rangle_j^{L-1} + u_j(\text{over } L-1)$

Explanation: The idea behind the conversion is that if the original shares of a over L does not wrap around the ring L, then it is also a valid shares of a over $L-1$. But if it does wrap around L, then we need to subtract 1 from shares of a over L to convert it to shares over $L-1$. Hence, $\theta = \texttt{wrap}(\langle a\rangle_0^L, \langle a\rangle_1^L, L)$ is introduced as the wrap bit which tells us whether shared of a wraps around L or not. Once θ is known, shares of a over $L-1$ is calculated by subtracting shares of θ over $L-1$ from shares of a over L as shown in step 11 above.

Now, to find θ, we can not simply exchange shares of a between P_0 and P_1 and check whether a wraps around L or not - that would reveal what a is. Hence, they mask the shares of a with shares of a common random number $r \in L$ before exchanging the shares of $x = a + r$. Then, for shares of x too, we need to find out whether x wraps around L or not. Let η be the wrap bit for the sum x. It can be computed by comparing $x > r-1$ using $\mathcal{F}_{PC}$; if x doesn't wrap around L, then $x > r-1$ and $\eta = 0$, else x wraps around L which means $x \not> r-1$ and $\eta = 0$.

As mentioned in the paper, the following relations over the integers hold by the correctness of `wrap` which will enable us to find the shares of θ:

i) $r = \langle r\rangle_0^L + \langle r\rangle_1^L - \alpha L$

ii) $\langle \tilde{a}\rangle_j^L = \langle a\rangle_j^L + \langle r\rangle_j^L - \beta_j L$

iii) $x = \langle \tilde{a}\rangle_0^L + \langle \tilde{a}\rangle_1^L - \delta L$

iv) $x = a + r - (1-\eta)L$

v) $a = \langle a\rangle_0^L + \langle a\rangle_1^L - \theta L$

Computing (i) - (ii) - (iii) + (iv) + (v), we get:

$$\begin{aligned} r - \langle\tilde{a}\rangle_0^L - \langle\tilde{a}\rangle_1^L - x + x + a &= \langle r\rangle_0^L + \langle r\rangle_1^L - \alpha L - \langle a\rangle_0^L - \langle r\rangle_0^L + \beta_0 L - \langle a\rangle_1^L - \langle r\rangle_1^L + \beta_1 L \\ &\quad - \langle\tilde{a}\rangle_0^L - \langle\tilde{a}\rangle_1^L - \delta L + a + r - (1-\eta)L + \langle a\rangle_0^L + \langle a\rangle_1^L - \theta L \\ r - \langle\tilde{a}\rangle_0^L - \langle\tilde{a}\rangle_1^L + a &= -\alpha L + \beta_0 L + \beta_1 L - \langle\tilde{a}\rangle_0^L - \langle\tilde{a}\rangle_1^L + \delta L + a + r - (1-\eta)L - \theta L \\ 0 &= -\alpha L + \beta_0 L + \beta_1 L + \delta L - (1-\eta)L - \theta L \\ 0 &= -\alpha + \beta_0 + \beta_1 + \delta - 1 + \eta - \theta \\ \theta &= \beta_0 + \beta_1 - \alpha - 1 + \delta + \eta \end{aligned} \tag{19}$$

In the protocol, step 10 calculates θ using the relation 19 which is then used in step 11 to compute shares of a over $L-1$.

3.2.4. Compute MSB

Compute MSB ($\mathcal{F}_{\text{MSB}}$) provides the functionality to compute the MSB of a value in the odd ring $\mathbb{Z}_{L-1}$. At the start of the protocol, P_0 and P_1 hold shares of a over odd ring $\mathbb{Z}_{L-1}$ and end up wit shares of $\alpha = \texttt{MSB}(a) = \texttt{LSB}(y = 2a)$. The protocol makes two calls: one to $\mathcal{F}_{\text{PC}}$, and the other to $\mathcal{F}_{\text{MatMult}}$.

Algorithm 6 Compute MSB $\prod_{\text{MSB}}(\{P_0, P_1\}, P_2)$

Input: P_0 and P_1 hold $\langle a\rangle_0^{L-1}$ and $\langle a\rangle_1^{L-1}$ respectively.
Output: P_0 and P_1 get $\langle\texttt{MSB}(a)\rangle_0^L$ and $\langle\texttt{MSB}(a)\rangle_1^L$ respectively.
Common Randomness: P_0 and P_1 hold a random bit β and random shares of 0 over L, denoted by u_0 and u_1 respectively.

1: P_2 picks $x \xleftarrow{\$} \mathbb{Z}_{L-1}$. Next P_2 generates $\langle x\rangle_j^{L-1}$, $\{\langle x[i]\rangle_j^p\}_{i\in[l]}$ and $\langle x[0]\rangle_j^L$ for $j \in \{0,1\}$ and sends to P_j
2: For $j \in \{0,1\}$, P_j computes $\langle y\rangle_j^{L-1} = 2\langle a\rangle_j^{L-1}$ and $\langle r\rangle_j^{L-1} = \langle y\rangle_j^{L-1} + \langle x\rangle_j^{L-1}$
3: P_0 and P_1 reconstructs r by exchanging shares.
4: P_0, P_1, P_2 call $\prod_{PC}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$, having input $\Big(\{\langle x[i]\rangle_j^p\}_{i\in[l]}, r, \beta\Big)$ and P_2 learns β'.
5: P_2 generates $\langle\beta'\rangle_j^L$ and sends to P_j for $j \in \{0,1\}$
6: For $j \in \{0,1\}$, P_j executes Steps 7-8.
7: $\langle\gamma\rangle_j^L = \langle\beta'\rangle_j^L + j\beta - 2\beta\langle\beta'\rangle_j^L$
8: $\langle\delta\rangle_j^L = \langle x[0]\langle_j^L + jr[0] - 2r[0]\langle x[0]\rangle_j^L$
9: P_0, P_1, P_2 call $\prod_{\text{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle\gamma\rangle_j^L, \langle\delta\rangle_j^L)$ and P_j learns $\langle\theta\rangle_j^L$.
10: For $j \in \{0,1\}$, P_j outputs $\langle\alpha\rangle_j^L = \langle\gamma\rangle_j^L + \langle\delta\rangle_j^L - 2\langle\theta\rangle_j^L + u_j$

Explanation: As discussed in the section 3.1, SecureNN changes computation of $\texttt{MSB}(a)$ problem to computation of $\texttt{LSB}(y = 2a) = y[0]$. Then, with addition of mask x to y such that $r = y + x$, it suffices to check whether the sum r wraps around $\mathbb{Z}_{L-1}$ or not. In other words, wrap $(y, x, L-1) = \gamma = (x > r)$ is gives the wrap bit and $\texttt{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus \gamma$. When the sum r wraps around the ring, its parity changes as (mod $n = 2^64 - 1$) is applied on it. Let's try to understand how that works in an odd ring $\mathbb{Z}_7$.

Case 1: $r = y + x$ doesn't wrap around $\mathbb{Z}_{l-1} \implies \gamma = 0$

We take $a = 1$ and $\texttt{MSB}(a) = 0 \implies y = 2a = 2$ and since y is always even, $\texttt{LSB}(y) = y[0] = 0$.
[1mm] Then, we have two cases:

i) When x is even, $x = 4$

$$\begin{aligned} r &= x + y \pmod 7 \\ &= 4 + 2 && \text{(even + even)} \\ &= 6 \pmod 7 \\ &= 6 && \text{(even)} \end{aligned}$$

So, the final r is even and $r[0] = 0$, and hence, $y[0] = r[0] \oplus x[0] \oplus \gamma = 0 \oplus 0 \oplus 0 = 0$

ii) When x is odd, $x = 3$

$$\begin{aligned} r &= x + y \pmod 7 \\ &= 3 + 2 && \text{(odd + even)} \\ &= 5 \pmod 7 \\ &= 5 && \text{(odd)} \end{aligned}$$

So, the final r is odd and $r[0] = 1$, and hence, $y[0] = r[0] \oplus x[0] \oplus \gamma = 1 \oplus 1 \oplus 0 = 0$

Hence, the parity of final r doesn't flip when $\gamma = 0$.

Case 2: $r = y + x$ doe wrap around $\mathbb{Z}_{l-1} \implies \gamma = 1$
We take $a = 2$ and $\texttt{MSB}(a) = 0 \implies y = 2a = 4$ and since y is always even, $\texttt{LSB}(y) = y[0] = 0$.
Then, we have two cases:

i) When x is even, $x = 6$

$$\begin{aligned} r &= x + y \pmod 7 \\ &= 6 + 4 && \text{(even + even)} \\ &= 10 \pmod 7 \\ &= 3 && \text{(odd)} \end{aligned}$$

So, the final r is odd and $r[0] = 1$, and hence, $y[0] = r[0] \oplus x[0] \oplus \gamma = 1 \oplus 0 \oplus 1 = 0$

ii) When x is odd, $x = 5$

$$\begin{aligned} r &= x + y \pmod 7 \\ &= 5 + 4 && \text{(odd + even)} \\ &= 9 \pmod 7 \\ &= 2 && \text{(even)} \end{aligned}$$

So, the final r is even and $r[0] = 0$, and hence, $y[0] = r[0] \oplus x[0] \oplus \gamma = 0 \oplus 1 \oplus 1 = 0$

Hence, the parity of final r does flip when it wraps around the $\mathbb{Z}_{L-1}$ and we introduce the wrap bit γ to negate the change.

In both the example above, we chose a such that they represented positive numbers in the

$\mathbb{Z}_7$ but a can chosen such that its represents a negative number like $a > \frac{n}{2}$ in the $\mathbb{Z}_{L-1}$. In that case, the $\texttt{MSB}(a) = \texttt{LSB}(y)$ is preserved but the parity of y changes. For an odd y the two cases shown above can be carried about similarly to see how the parity of sum r changes when it wraps.
Irrespective of what x and y are, the end case is that r can either be odd or even, and when it wraps around the $\mathbb{Z}_{L-1}$, the (mod $2^{64} - 1$), an odd number, operation will change the parity of r and hence, wrap bit γ is introduced to negate the change and find the $\texttt{LSB}(y)$.

3.3. Main Protocols

3.3.1. DReLU

DReLU ($\mathcal{F}_{\text{DReLU}}$) provides the functionality to compute the derivative of ReLU denoted mathematical by ReLU'. It is a 3-party protocol where P_0 and P_1 start with secret shares of a over ring $\mathbb{Z}_L$ and at the end get the shares of $\epsilon =$ ReLU'(a) over $\mathbb{Z}_L$. ReLU'$(a) = 1$ iff MSB$(a) = 0$, else ReLU'$(a) = 0$. The protocol makes two calls: one to $\mathcal{F}_{\text{SC}}$, and the other to $\mathcal{F}_{\text{MSB}}$.

Algorithm 7 ReLU' $\prod_{\text{DReLU}}(\{P_0, P_1\}, P_2)$

Input: P_0 and P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$ respectively.
Output: P_0 and P_1 get $\langle \text{ReLU'}(a) \rangle_0^L$ and $\langle \text{ReLU'}(a) \rangle_1^L$ respectively.
Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L$ denoted by u_0 and u_1 respectively.

1: For $j \in \{0, 1\}$, parties P_j computes $\langle c \rangle_j^L = 2\langle a \rangle_j^L$.
2: P_0 , P_1 and P_2 run $\prod_{\text{SC}}(\{P_0, P_1\}, P_2)$ with P_0 and P_1 having inputs $\langle c \rangle_0^L$ and $\langle c \rangle_1^L$ respectively, and P_0 and P_1 learn $\langle y \rangle_0^{L-1}$ and $\langle y \rangle_1^{L-1}$ respectively.
3: P_0 , P_1 and P_2 run $\prod_{\text{MSB}}(\{P_0, P_1, P_2)$ with P_0 and P_1 having inputs $\langle y \rangle_0^{L-1}$ and $\langle y \rangle_1^{L-1}$ respectively, and P_0 and P_1 learn $\langle \alpha \rangle_0^L$ and $\langle \alpha \rangle_1^L$ respectively.
4: For $j \in \{0, 1\}, P_j$ outputs $\langle \epsilon \rangle_j^L = j - \langle \alpha \rangle_j^L + u_j$.

Explanation: As outlined in the section 3.1, to compute ReLU'(a) , the protocol computes the MSB(a). To do so, it first converts the shares of a from over $\mathbb{Z}_L$ to over $\mathbb{Z}_{L-1}$ using $\mathcal{F}_{\text{SC}}$. For the correctness of the Share Convert functionality, the input value to $\mathcal{F}_{\text{SC}}$ must not be equal to $L - 1$ which is ensured by computing the shares of $c = 2a$ and fixing it as the input to $\mathcal{F}_{\text{SC}}$. In general, the value of input data to PPNN is assumed to be not larger than 2^k, where $k < l - 1$. This implies that if the value of a is such that $a \in [a, 2^k) \cup (2^l - 2^k, 2^l - 1]$, then ReLU'$(a) =$ ReLU'(c).

3.3.2. ReLU

ReLU ($\mathcal{F}_{\text{ReLU}}$) provides the functionality to compute ReLU(a). It is a 3-party protocol where P_0 and P_1 start with secret shares of a over ring $\mathbb{Z}_L$ and at the end get the shares of $c =$ ReLU'(a) over $\mathbb{Z}_L$. ReLU$(a) = 1$ iff MSB$(a) = 0$, else ReLU(a) = 0. Hence, ReLU$(a) =$ ReLU'$(a) \cdot a$. The protocol makes two calls: one to $\mathcal{F}_{\text{DReLU}}$, and the other to $\mathcal{F}_{\text{MatMult}}$.

Algorithm 8 ReLU $\prod_{\text{ReLU}}(\{P_0, P_1\}, P_2)$

Input: P_0 and P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$ respectively.
Output: P_0 and P_1 get $\langle \text{ReLU}(a) \rangle_0^L$ and $\langle \text{ReLU}(a) \rangle_1^L$ respectively.
Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L$ denoted by u_0 and u_1 respectively.

1: P_0 , P_1 and P_2 run $\prod_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ with P_0 and P_1 having inputs $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$ respectively, and P_0 and P_1 learn $\langle \epsilon \rangle_0^L$ and $\langle \epsilon \rangle_1^L$ respectively.
2: P_0, P_1 and P_2 run $\prod_{\text{MatMult}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $(\langle \epsilon \rangle_j^L, \langle a \rangle_j^L)$ and P_0 and P_1 learn $\langle c \rangle_0^L$ and $\langle c \rangle_1^L$ respectively.
3: For $j \in \{0, 1\}, P_j$ outputs $\langle c \rangle_j^L + u_j$.

Explanation: As mentioned above, `ReLU(`a`) = ReLU'(`a`)` $\cdot\ a$. Hence, `ReLU'(`a`)` is computed by invoking $\mathcal{F}_{\text{DReLU}}$ and then the result is multiplied with a using $\mathcal{F}_{\text{MatMult}}$ to get `ReLU(`a`)`.

3.3.3. Division

Division provides ($\mathcal{F}_{\text{DIV}}$) provides the functionality of securely dividing two numbers in a ring between two parties, P_0 and P_1. P_0 and P_1 start with the shares of x and y over $\mathbb{Z}_L$, and at the end get the shared of $\lfloor x/y \rfloor$ over $\mathbb{Z}_L$ where $y \neq 0$. The protocol makes two calls: one to $\mathcal{F}_{\text{DReLU}}$, and the other to $\mathcal{F}_{\text{MatMult}}$.

Algorithm 9 Division $\prod_{\text{DIV}}(\{P_0, P_1\}, P_2)$

Input: P_0 and P_1 hold $(\langle x \rangle_0^L, \langle y \rangle_0^L)$ and $(\langle x \rangle_1^L, \langle y \rangle_1^L)$ respectively
Output: P_0 and P_1 get $\langle x/y \rangle_0^L$ and $\langle x/y \rangle_1^L$ respectively.
Common Randomness: P_0 and P_1 hold l shares of 0 over $\mathbb{Z}_L$, denoted by $w_{i,0}$ and $w_{i,1}$ for all $i \in [l]$ respectively. They additionally also hold another shares of 0 over $\mathbb{Z}_L$, denoted by s_0 and s_1.

1: Set $u_l = 0$ and for $j \in \{0,1\}, P_j$ holds $\langle u_l \rangle_j^L$ (through common randomness).
2: **for** $i = \{l-1, \ldots, 0\}$ **do**
3: $P_j, j \in \{0,1\}$, computes $\langle z_i \rangle_j^L = \langle x \rangle_j^L - \langle u_i + 1 \rangle_j^L - 2^i \langle u \rangle_j^L + w_{i,j}$.
4: P_0, P_1, P_2 call $\prod_{\text{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle z \rangle_j^L$, and P_0 and P_1 learn $\langle \beta_i \rangle_0^L$ and $\langle \beta_i \rangle_1^L$ respectively.
5: P_0, P_1, P_2 call $\prod_{\text{MatMult}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i \rangle_j^L, \langle 2^i y \rangle_j^L)$, and P_0 and P_1 learn $\langle v_i \rangle_0^L$ and $\langle v_i \rangle_1^L$ respectively.
6: $P_j, j \in \{0,1\}$, computes $\langle k \rangle_j^L = 2^i \cdot \langle \beta_i \rangle_j^L$.
7: $P_j, j \in \{0,1\}$, computes $\langle u_i \rangle_j^L = \langle u_{i+1} \rangle_j^L + \langle v_i \rangle_j^L$.
8: **end for**
9: For $j \in \{0,1\}, P_j$ computes $\langle q \rangle_j^L = \sum_{i=0}^{l-1} \langle k_i \rangle + s_j$.

Explanation: The protocol implements normal long division where the quotient is computed bit-by-bit sequentially from `MSB` - that is it's binary representation is computed bit-by-bit starting form `MSB`. For each iteration $i \in l$, the current dividend (z_i) is computed by subtracting the correct multiple of the divisor ($2^i y$). To find the correct multiple of the divisor ($2^i y$), the difference of the current dividend (z_i) and multiple of the divisor ($2^i y$) is compared to 0 using $\mathcal{F}_{\text{DReLU}}$ functionality. Based on the result of the comparison, the i-th bit of the quotient is set to 0 or (2^i), and accordingly, the correct multiple of divisor to be subtracted from the dividend (z_i) is chosen between 0 and $2^i y$.

For instance, let's understand the protocol by an example. Let $x = 13 = 1101$ and $y = 5 = 101$. Here, $l = 4$, $i \to 3$ to 0, and $u_4 = 0$.

- When $i = 3, u_4 = 3$

  3. $z_3 = x - u_4 - 2^3 y = 13 - 0 - 2^3 \times 5 = 13 - 40 = -27$
  4. $\prod_{\text{DReLU}}$ gives $\beta_3 = 0$ as $z_3 < 0$. So, $2^3 \times 5$ is not the correct multiple to be subtracted.
  5. Hence, using $\prod_{\text{MatMult}}$, we get $v_3 = \beta_3 \times 2^3 y = 0 \times 2^3 \times 5 = 0$ which is the correct multiple to be subtracted.

6. Also, the 3^{rd} bit of the quotient is set 0, $k_3 = 2^3 \times \beta_3 = 2^3 \times 0 = 0$.
7. For the next iteration, $u_3 = u_4 + v_3 = 0 + 0 = 0$. So far, 0 has been subtracted from x.

- When $i = 2, u_3 = 0$

3. $z_2 = x - u_3 - 2^2 y = 13 - 0 - 2^2 \times 5 = 13 - 20 = -7$
4. $\beta_2 = 0$ as $z_2 < 0$. So, $2^2 \times 5$ is not the correct multiple to be subtracted.
5. Hence, $v_2 = \beta_2 \times 2^2 y = 0 \times 2^2 \times 5 = 0$ is the correct multiple to be subtracted.
6. Also, $k_2 = 2^2 \times \beta_2 = 2^2 \times 0 = 0$.
7. For the next iteration, $u_2 = u_3 + v_2 = 0 + 0 = 0$.

- When $i = 1, u_2 = 0$

3. $z_1 = x - u_2 - 2^1 y = 13 - 0 - 2^1 \times 5 = 13 - 10 = 3$
4. $\beta_1 = 1$ as $z_1 > 0$. So, $2^1 \times 5$ is the correct multiple to be subtracted.
5. Hence, $v_1 = \beta_1 \times 2^1 y = 1 \times 2^1 \times 5 = 10$.
6. Also, $k_1 = 2^1 \times \beta_1 = 2^1 \times 1 = 2$.
7. For the next iteration, $u_1 = u_2 + v_1 = 0 + 10 = 10$.

- When $i = 0, u_1 = 10$

3. $z_0 = x - u_1 - 2^0 y = 13 - 10 - 2^0 \times 5 = 13 - 15 = -2$
4. $\beta_0 = 0$ as $z_0 < 0$. So, $2^0 \times 5$ is not the correct multiple to be subtracted.
5. Hence, $v_0 = \beta_0 \times 2^0 y = 0 \times 2^0 \times 5 = 0$ is the correct multiple to be subtracted.
6. Also, $k_0 = 2^0 \times \beta_0 = 2^0 \times 0 = 0$.
7. For the next iteration, $u_0 = u_1 + v_0 = 10 + 0 = 10$.

Then, quotient $(q) = \sum_{i=0}^{l-1} k_i = k_0 + k_1 + k_2 + k_3 = 0 + 2 + 0 + 0 = 2$ and $\lfloor 13/5 \rfloor$ is also 2. So, this is how $\prod_{\text{DIV}}$ divides two numbers in the ring securely.

3.4. Building the PPNN Model

We now have the functionalities to implement the main three operations as mentioned in section 2.1.5.5 in privacy preserving manner to build a PPNN with ReLU and Softmax for solving a multi-class classification problem. In SOC setting, multiplication operation is securely realized by $\mathcal{F}_{\text{MatMult}}$, computation of ReLU is securely realized by $\mathcal{F}_{\text{ReLU}}$, division operation is securely realized by $\mathcal{F}_{\text{DIV}}$, and computation of Softmax is securely realized by computation of `ASM(·)` function[18] which uses both $\mathcal{F}_{\text{ReLU}}$ and $\mathcal{F}_{\text{DIV}}$ for its calculation. In essence, for a layer in NN, we have a fully connected layer, followed by ReLU or Softmax if the layer is the last layer. To implement this in privacy preserving manner in PPNN, we first use $\mathcal{F}_{\text{MatMult}}$, followed by $\mathcal{F}_{\text{ReLU}}$ or computation of `ASM(·)` if its the last layer.

Hence, for a PPNN, the forward propagation at each layer l which is given by the equations 10 and 11 transforms to, for parties $P_s, s \in \{0, 1\}$, as:

$$\langle z_j^l \rangle_s^L = f\left(\sum_{i=0}^{d_{l-1}} \prod_{\text{MatMult}} (\langle w_{ji} \rangle_s^L \times \langle z_i^{l-1} \rangle)\rangle_s^L\right) \tag{20}$$

where f is computed as `ReLU` or `ASM` based on which layer it is.

Similarly, the backward propagation at each layer l which is given by equation 17 transforms to, for parties $P_s, s \in \{0, 1\}$, as:

$$\langle w_{ji}^l \rangle_s^L \ (new) := \langle w_{ji}^l \rangle_s^L \ (old) - \prod_{\text{MatMult}} \left(\alpha \times \left\langle \frac{\partial J}{\partial w_{ji}^l} \right\rangle_s^L\right) \tag{21}$$

where $\left\langle \frac{\partial J}{\partial w_{ji}^l} \right\rangle_s^L$ is computed in privacy preserving manner using the relations obtained from Delta rule as shown in section 2.1.5.4.

References

[1] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-46766-1_34.

[2] Ivan Damgard, Martin Geiseler, and Mikkel Kroigard. Homomorphic encryption and secure comparison. *Int. J. Appl. Cryptol.*, 1(1), 2008. https://doi.org/10.1504/IJACT.2008.017048.

[3] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. https://doi.org/10.1109/MSP.2012.2211477.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, Cambridge, Massachusetts, 2017. https://dl.acm.org/doi/book/10.5555/3086952.

[5] Yehuda Lindell and Ariel Nof. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276, 2017. https://doi.org/10.1145/3133956.3133999.

[6] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 591–602, 2015. https://doi.org/10.1145/2810103.2813705.

[7] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017. https://doi.org/10.1109/SP.2017.12.

[8] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography – PKC 2007*, pages 343–360, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-71677-8_23.

[9] Arpita Patra and Ajith Suresh. Blaze: Blazing fast privacy-preserving machine learning. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. http://dx.doi.org/10.14722/ndss.2020.24202.

[10] Kuber Shahi. Implementation Link. https://github.com/kubershahi/ashoka-capstone.

[11] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. In *Proceedings on Privacy Enhancing Technologies*, volume 3, pages 26–49, 2019. https://doi.org/10.2478/popets-2019-0035.