IT UNIVERSITY OF CPH

IT University of Copenhagen

Master Thesis

Privacy Preserving Machine Learning

KISPECI1SE

Authors:
Emil Christian Hørning (echh@itu.dk)
Alexander Albert Ramos (aara@itu.dk)

Supervisor:
Bernardo Machado David

June 2, 2020

Abstract

This is a report conducted in relation to our thesis from ITU in Spring 2020. The thesis is a collaborate work between two students with different specializations; *security* and *machine learning*. In this thesis we describe and evaluate and end-to-end protocol for doing privacy preserving machine learning presented in the research paper *'SecureNN: 3-Party Secure Computation for Neural Network Training' by Wagh, Gupta, and Chandran* [1], and provide improvements from another paper *'High Performance Logistic Regression for Privacy-Preserving Genome Analysis' by De Cock, Dowsley, Nascimento, Railsback, Shen, Todoki* [2]. We first provide background knowledge from security and machine learning perspectives on privacy preserving machine learning, then we attempt to simplify and describe the methods used in [1] and [2]. Furthermore we implement the protocols in python 3.7 and provide data on execution time of the analyzed subprotocols, finally we discuss how to improve [1] using methods from [2].

Contents

1 Introduction

Within the field of machine learning, (deep) neural networks have proven to be a powerful tool in creating predictive models for a range of applications, such as business analysis, healthcare, image classification, game AI, and so on. Their predictive performance heavily relies on data which is used to train a model, and such data can be obtained either from a single contributors or multiple contributors. However this data might be sensitive (e.g. medical - or financial records), and can't be revealed in clear text to any other contributor or training entity due to compliance requirements. This puts a lot of unwanted trust in any party involved in the process, from training the neural network to querying it. Therefore, in order to train a neural network model with good accuracy, it becomes highly desirable to securely train over data, so that any plain-text data is kept hidden from all parties outside the contributor of the data.

A solution that can fulfill such requirements can be implemented using multi-party computation (MPC) together with secret shares, which is the approach we will investigate in this report. Using MPC it's possible to create a system using N distributed parties, that given data from one or more contributors as input, can follow an interactive protocol and perform necessary machine learning computations which produce satisfyingly accurate models, while never revealing any of the original input. The model can subsequently be queried using new inputs to obtain predictions/inference. Similarly, any trained model can be retained as secret shares, effectively keeping its parameters hidden, thus preventing malicious actors from learning anything about the model and hereby exploiting/copying it. An example of a desirable solution is described in *SecureNN*, and looks as such:

1. A group of M hospitals each currently holding sensitive data (e.g. medical records containing bodyfat, blood group, heart rate, ect.), are interested in training a model so that given a future patient's medical record as input, the model can predict whether that patient suffers from some disease or not.

2. The M hospitals each submit their data as secret shares to the N-party system, which trains a model adhering to its MPC protocols until a satisfactory model is obtained.

3. The system can then use its trained model to run prediction as a service, which any hospital can query by submitting a new patients medical record as input, for which it receives an output predicting whether that patient suffers from a disease or not.

4. The complete system can be set up in such a way that the medical records used for training, any new input, and any predicted output only remains visible to the querying party, hereby assuming no trust between involved parties.

Figure 1: Example of a secure ML system from SecureNN [1]

This paper will describe and reflect on the methods used in the research paper *'SecureNN: 3-Party Secure Computation for Neural Network Training' by Wagh, Gupta, and Chandran* [1] (*SecureNN*), which proposes a 3-party computation protocol for neural network training. Furthermore, we will analyze the results published in *SecureNN* and attempt to identify potential bottlenecks[1]. Additionally we will describe an alternative approach used by another paper *'High Performance Logistic Regression for Privacy-Preserving Genome Analysis' by De Cock et. al* [2] (*The Genome Paper*) which uses a 2-party protocol to perform fast and secure logistic regression. Finally, we will describe our experiments comparing the protocols of these two approaches, using our own implementation, report our findings and discuss the results.

[1]Part of this analysis is taken from our Thesis Research Project on SecureNN from Autumn 2019

2 Background theory

2.1 Technical glossary

- ML: Machine Learning
- PPML: Privacy Preserving Machine Learning
- MPC: Multi Party Computation
- $\mathbb{Z}$: The set of integers
- λ: Bit length of numerical values used in a protocol, in this report $\lambda = 64$
- L: The ring size 2^λ in this report $L = 2^{64}$
- $\mathbb{Z}_L$: Mathematical ring of integers in range 0 to L (L excluded)
- $\mathbb{Z}_{L-1}$: Mathematical ring of integers in range 0 to L-1 (L-1 excluded)
- P_0, P_1, P_2: Individual parties participating in an interactive protocol
- p: a prime number
- MSB: Most Significant Bit
- LSB: Least Significant Bit
- Adversary: A party trying to learn the secrets in the protocol.
- Trusted initializer: A trusted party that provides common randomness to the working parties before the protocol starts.
- UC: Universal Composability - A model for analysing the security of cryptographic protocols. Proposed by Ran Canetti[5]. Authors of protocols claim them to be secure in reference to UC - UC Secure.

2.2 Secure Multi-Party Computation

Secure Multi-Party Computation is a field within cryptography where the goal is to create methods for multiple parties to jointly compute a function over their inputs, while keeping those inputs private to the parties themselves, and eavesdroppers on the network.
The goals that we want to achieve using Secure Multi-Party Computations are:

- **Input Privacy.** No actors in the protocol will be able to infer information about the data from the messages being passed around during the protocol.
- **Correctness** Any number of colluding dishonest parties participating in the protocol, should not be able to, by deviating from the protocol, force or trick an honest party to output something that is wrong. This is achieved by either having a *robust* protocol that always gives the correct output, or having a protocol: *MPC with abort functionality*, which will stop the protocol once it is evident something is wrong.

Keeping inputs secret is achieved using additive secret-shares, a cryptographic primitive that allows for values to be split into secrets *s0, s1* such that only by combining *s0* and *s1* an actor may know the value of the original secret.

2.3 Adversarial model

When talking about security in multiparty computations it is good to define what an adversary is capable of, and which measures are taken to counter an adversary. In this paper we note two different adversarial capabilities; *Passive* and *Active*, and how they impact *input privacy* and *correctness*.

Passive adversary: A passive adversary is defined as an adversary that corrupts one of the three actors and rightfully follows the protocol, but tries to learn information from the protocol. The authors of *SecureNN* claims full input privacy and correctness under a single corrupted passive process. Input privacy follows by a security proof, proving indistinguishability between *real* and *ideal* functionality for an environment `Z` with any input. This simulation argument is expanded upon in [5] and [6] . The correctness trivially follows, since the definition of passive adversaries include that they dutifully follows the protocol.

Active adversary: An active adversary is an adversary that corrupts one of the three actors and may arbitrarily deviate from the protocol, in order to tamper with the output (correctness), or learn about the input (privacy). The authors of *SecureNN* claim privacy against an active adversary that corrupts any of the three parties, the argument is based on indistinguishability between any two inputs from honest parties. Correctness however cannot be achieved so any active adversary that corrupts a single party may have the protocol conclude with incorrect output and the honest parties will never notice.

2.4 Deep neural networks

Deep neural networks (DNN) have proven to be exceptionally useful in solving prediction and classification problems on data with high dimensional input, which displays non-linear separability. The key idea behind a neural network, is to approximate the non-linear function which describes the data - this process is referred to as *training*. Once the function has been approximated, it can be used to predict some output given an input - this part is referred to as *testing/prediction/inference*. Training a network requires a large data set from which the network learn some arbitrary pattern, and adjusts its parameters (also referred to as weights and biases). The structure of a network is divided into layers, where a layer consists of nodes (neurons), edges (weights) and biases. Each layer is fully connected to the following layer in the network (all nodes in one layer have an edge to all nodes in the next layer). The network consists of an input layer, an output layer, and one or more hidden layers in between. For classification with more than one class, it is common to use a *Softmax* activation function in the output layer. The *Softmax* layer normalizes output values to a probability distribution between the K classes of the output. In other words, the network outputs a confidence in its prediction [7]. A general Neural Network

structure can be seen in figure 2.

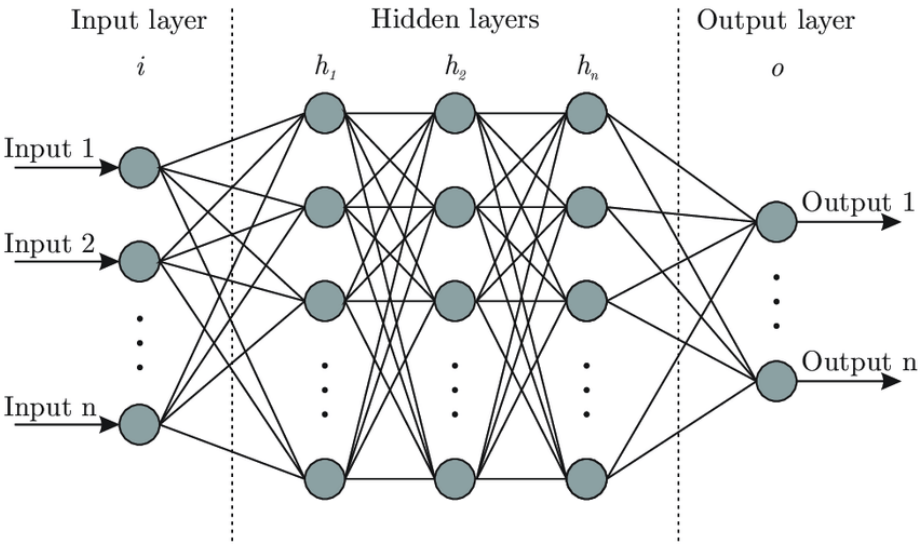

Figure 2: Architecture of a neural network

Each node in a hidden (and output) layer takes as input the weighted sum of outputs from the previous layer and applies an activation function to it, which outputs a value which is then used as an input in the subsequent layer. The activation function has the purpose of introducing non-linearity to the network, hereby allowing it to approximate a non-linear function. This process of computing the weighted sums for all nodes in a layer, can simplified as a matrix multiplication between an input matrix X and a weight matrix W. So the entire forward propagation essentially just involves performing matrix multiplication and then applying some activation function to that result, iteratively until a final output is produced at the last layer. Training a network involves three main steps:

1. Feeding the input forward through the network which is initialized with random weights, producing some output

2. Computing the loss of the produced output using a loss-function. That is, how far the produced output is from the true label (value it should be, according to the training data)

3. Adjusting all weights and biases throughout the network with respect to the computed loss using an algorithm called back-propagation, ultimately minimizing loss.

When a network has been trained sufficiently, it can be given new input which it simply feeds forward through the network as in step 1 and produces an output [8].

2.5 Secure bit-decomposition of secret shared integers

In MPC, getting shares of the bit-decomposition of a secret-shared value can provide important knowledge. We know that the MSB of an integer is used to denote the signing (whether it is positive or negative), while the LSB tells us if the integer is even or odd. In MPC when we want to achieve bit-decomposition we have two parties P_0 and P_1 holding additive shares of some integer S, denoted as S_0 and S_1, at the end of the bit-decomposition protocol we want P_0 and P_1 to hold XOR sharings of the secret S, denoted as S_0^i and S_1^i, with $i = 1, 2, ..., \lambda$ (λ = length of bit string) such that the bit string of S referred to as S_i may be reconstructed by XOR'ing the bit-wise shares S_0^i and S_1^i held by P_0 and P_1. Trivially from this we also see that P_0 and P_1 hold shares of specific bits from the secret value.

In XOR sharings every bit is in Z_2 for which some interesting modular arithmetic is observed. We know that the integers we have in $\mathbb{Z}_2$ are only 1 and 0. This results in the following equivalences to bitwise operators.

Multiplication becomes bitwise AND.

a	b	$a \cdot b$
1	1	1
0	1	0
1	0	0
0	0	0

Addition becomes bitwise XOR

a	b	$a + b$
1	1	0
0	1	1
1	0	1
0	0	0

With this in mind, it becomes easier to understand the bit-decomposition protocols described in this paper.

3 SecureNN

SecureNN [1] presents a 3 party protocol for realizing secure computation during neural network training/inference. Their approach uses 3 parties P_0, P_1, and P_2 which together run a set of interactive protocols. There are two subsets of protocols:

1. *Supporting protocols* which are Matrix Multiplication, Select Share, Private Compare, Share Convert, Compute MSB. These protocols ensure secure computation of "trivial" functionality.

2. *Main protocols* which are linear/convolutional layer, Derivative of Relu, Relu, Division, Maxpool, Derivative of Maxpool. These realize the higher order functionality of a (convolutional) neural network.

The main protocols uses the supporting protocols to realize their own functionality, and finally an end-to-end protocol can be constructed by making consecutive calls to these main protocols resulting in secure neural network training. Each of these protocols are described below.

3.1 Supporting protocols

3.1.1 Matrix Multiplication

The Matrix Multiplication protocol (`MatMul`) allows for P_0 and P_1, each holding shares of matrix X and Y, to securely multiply the matrices X and Y so that by the end of the protocol each process holds shares of $X \cdot Y$, this is done with the aid of P_2. At no point does either process see X, Y, or $X \cdot Y$ in clear. This is done using beaver triplets [9] generated by P_2. These triplets are then split into secrets and given to P_0 and P_1. P_0 and P_1 then mask their X and Y shares using the triplets, then share these new masked values and can finally reconstruct each share of $X \cdot Y$. The shares of this final value has the common randomness u added to it, allowing no information to leak.

Input: P_0 and P_1 each hold shares of X and Y and shares of a zero matrix U.
Output: P_0 and P_1 each get shares of $X \cdot Y$.

1. P_2 creates random matrices A and B and calculates C such that $A \cdot B = C$. P_2 creates two shares of each of these values and send the shares of each value to their respective parties P_0 and P_1.

2. P_0 and P_1 each compute their respective shares of E and F, such that $E = X - A$ and $F = Y - B$, they then reconstruct E and F by exchanging the shares.

3. Finally P_0 and P_1 outputs shares of $X \cdot Y$ as $-jE \cdot F + X_j \cdot F + E \cdot Y_j + C_j + U_j$ where $j \in \{0, 1\}$

3.1.2 Select Share

The Select Share protocol (`SS`) allows for P_0 and P_1 to get new shares of either x or y, without the two parties knowing which they get. At first the parties

hold their shares of x and y and shares of a random bit **0** or **1** called α over the ring $\mathbb{Z}_L$.
First P_0 and P_1 compute shares of w such that $w = x - y$. Each party then use `MatMul` to compute shares of c, such that $c = w \cdot \alpha$. Following this, each of the parties P_0 and P_1 compute shares of z such that $z = x + c + u$.
Opening this up we see that we get $x + \alpha \cdot (y - x)$ which intuitively is the selection of x if $\alpha = 0$ or y if $\alpha = 1$.

Input: P_0 and P_1 each hold shares of x, y, α and shares of a zero mask u.
Output: P_0 and P_1 each get shares of z where $z = x + \alpha \cdot (y - x)$.

1. P_0 and P_1 each compute shares of w such that $w = y - x$.
2. P_0 and P_1 in collaboration with P_2 perform `MatMul` to compute shares of c such that $c = w \cdot \alpha$.
3. Finally P_0 and P_1 outputs shares of z as $x_j + c_j + u_j$ where $j \in \{0, 1\}$

3.1.3 Private Compare

Private Compare (`PC`) secretly compares two values, x and r, outputting a boolean value denoting whether $(x > r)$, or not. P_0 and P_1 holds shares of a λ-bit integer x in a field $\mathbb{Z}_p$ ($p = 67$), a λ-bit integer r, and a bit β. P_2 learns at the end of the protocol a bit $\beta' = \beta \oplus (x > r)$ where $(x > r)$ denotes the bit being 1, if $x > r$ and 0 otherwise. P_0 and P_1 also hold λ common values $s_i \in \mathbb{Z}_p$ for all i $\in [\lambda]$ and a random permutation π for λ elements. Furthermore, P_0 and P_1 hold λ common random values $u_i \in \mathbb{Z}_p$.

Input: P_0 and P_1 hold shares of λ-bit integers of x: $\{x[i]_0^p\}$ and $\{x[i]_1^p\}$. They also both hold a common input r as a λ-bit representation, and a common bit β.
Output: P_2 gets a bit $\beta \oplus (x > r)$.

1. Initialize $t = r + 1 \ mod \ 2^l$
2. P_0 and P_1 (denoted $j \in \{0, 1\}$) **execute** steps 3-13:
3. **for** each i $= \{\lambda - 1, \lambda - 2, ..., 0\}$ **execute** steps 4-12:
4. **if** β is 0 **then**
5. Compute share j of w_i such that $w_i = x[i] + jr[i] - 2r[i]x[i]$
6. Compute share j of c_i such that $c_i = jr[i] - x[i] + j + \sum_{k=i+1}^{\lambda} w_k$
7. **else if** β is 1 AND $r \neq 2^\lambda - 1$ **then**
8. Compute share j of w_i such that $w_i = x[i] + jt[i] - 2t[i]x[i]$
9. Compute share j of c_i such that $c_i = -jt[i] + x[i] + j + \sum_{k=i+1}^{\lambda} w_k$
10. **else** (corner case where $r = 2^\lambda - 1$)
11. if $i \neq 1$: Compute share j of c_i such that $c_i = (1 - j)(u_i + 1) - ju_i$

12. else: Compute share j of c_i such that $c_i = -1^j \cdot u_i$

13. P_0 and P_1 send their respective shares of d_i to P_2 where $d_i = \pi(s_i c_i)$

14. P_2 computes for all $i \in [\lambda]$, the reconstruction of d_i and sets $\beta' = 1$ if there exists an i for which $d_i = 0$

15. P_2 outputs β'

3.1.4 Share Convert

Share Convert (`SC`) is a protocol to convert shares over $\mathbb{Z}_L$ to shares over $\mathbb{Z}_{L-1}$, where $L = 2^{64}$. P_0 and P_1 hold shares of a in $\mathbb{Z}_L$ where $a \neq L-1$. At the end of the protocol, P_0 and P_1 hold fresh shares of the same value over L-1, that is a^{L-1}. This protocol use a wrap-around function $\kappa := wrap(x, y, L)$ which is 1 if $x + y \geq L$ and 0 otherwise. In other words, κ denotes the wrap-around bit for $x + y$ mod L. That is, if the original shares of a wrapped around L ($\kappa = 1$), we need to subtract 1, else the original shares are also valid shares of the same value of a over L-1.

Input: P_0 and P_1 hold shares of a over the ring $\mathbb{Z}_L$ such that reconstructing these shares does not equal $L - 1$
Output: P_0 and P_1 get shares of a over $\mathbb{Z}_{L-1}$
Common randomness: P_0 and P_1 hold a random bit η'', a random r in $\mathbb{Z}_L$, shares of this random value r and shares of 0, denoted as u.

1. P_0 and P_1 each compute shares of $\tilde{a}$ and β such that $\tilde{a} = a + r$, and $\beta =$ `wrap(a,r,L)`. They then send their shares of $\hat{a}$ to P_2

2. P_2 then computes x such that x is the reconstruction of $\tilde{a}$ from its shares, and δ is the result of calling `wrap` with the two shares of $\tilde{a}$ and L.

3. P_2 then creates shares of the bit representation in $\mathbb{Z}_p$ of x and shares of δ and sends them to P_0 and P_1.

4. Private Compare (`PC`) is then called to compare the bit representation of x and *r-1*, η'' is the value the result gets XOR'ed with, to reveal η' to P_2.

5. P_2 then creates shares of η' which is sent to P_0 and P_1

6. P_0 and P_1 computes the next 3 steps where $j \in \{0, 1\}$.

7. $\eta_j = \eta'_j + (1 - j)\eta'' - 2\eta''\eta'_j$

8. $\theta_j = \beta'_j + (1 - j) \cdot (-\alpha - 1) + \delta_j + \eta_j$

9. Output $y_j = a_j - \theta_j + u_j$

3.1.5 Compute MSB

Compute MSB (`ComputeMSB`) is the function in the protocol to compute shares of the leftmost bit of some secret shared value a. It is worth noting that when the shares of a are over an odd ring $\mathbb{Z}_{L-1}$ then the `MSB` computation can be

converted into an LSB computation: MSB(*a*) = LSB(*2a*)

Input: P_0 and P_1 hold shares of a over the ring $\mathbb{Z}_{L-1}$
Output: P_0 and P_1 get shares of the most significant bit from a over the ring $\mathbb{Z}_L$.

1. P_2 picks a random x over the ring $\mathbb{Z}_{L-1}$ and then creates shares of the value, shares of the bit representation and shares of the least significant bit of x. These shares are then sent to P_0 and P_1.

2. P_0 and P_1 both compute shares of y over $\mathbb{Z}_{L-1}$ such that $y = 2a$. They also compute r over $\mathbb{Z}_{L-1}$ such that $r = y + x$ where x is their respective share received from P_2.

3. P_0 and P_1 reconstruct r by exchanging shares.

4. P_0, P_1, and P_2 compute the private comparison PC of the bit-representation of x and r. P_2 then learns β', which P_2 converts to shares and sends to P_0 and P_1.

5. P_0 and P_1 then compute the equation $r[0] \oplus x[0] \oplus (x > r)$ by using the arithmetic equation for xor computation; $x \oplus r = x + r - 2xr$. If any of x or y is public and known to both computing parties, this computation can be done over shares locally, otherwise if private using MatMul.

3.2 Main protocols

3.2.1 Linear and Convolutional layer

As described in 2.4 (*Deep neural networks*), a fully connected layer in a neural network, can be computed as a matrix multiplication between an input matrix X and a weight matrix W. In *SecureNN* they use Convolutional Neural Networks since they have proven to be highly efficient for image recognition. CNN's have convolutional layers instead of fully connected layers, which is a convolution of several filters over an input matrix [10]. However it is possible to express such a convolutional layer as a (larger) matrix multiplication allowing the use of the same supporting protocol MatMul. For example a 2D convolution of a 3x3 input matrix X with a filter K of size 2x2 can be represented as the following matrix multiplication:

$$\text{Conv2d}\left(\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix}\right) = \begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \\ x_4 & x_5 & x_7 & x_8 \\ x_5 & x_6 & x_8 & x_9 \end{bmatrix} \times \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix}$$

Figure 3: 2D convolution expressed as matrix multiplication

3.2.2 Derivative of ReLU

The algorithm Derivative of ReLU (`DReLU`) computes the derivative of `ReLU`. Parties P_0 and P_1 hold shares of a over $\mathbb{Z}_L$ and end up with secret shares of `RELU'(a)` over $\mathbb{Z}_L$. `RELU'(a)` $= 1$ if `MSB(a)` $= 0$, else `ReLU'(a)` $= 0$.
The protocol computes the shares of `MSB(a)` and flips it to obtain `ReLU'(a)`. The `MSB` protocol needs shares of a over $\mathbb{Z}_{L-1}$. Thus we first need to convert the shares of a over $\mathbb{Z}_L$ to fresh shares over $\mathbb{Z}_{L-1}$, this is done using the Share Convert protocol `SC`.

Input: P_0 and P_1 hold shares of a over the ring $\mathbb{Z}_L$
Output: P_0 and P_1 get shares of `ReLU'(a)`

1. P_0 and P_1 each compute shares of c, such that $c = 2a$ over the ring $\mathbb{Z}_L$
2. P_0 and P_1 obtains y by converting the shares of c in $\mathbb{Z}_L$ to shares in $\mathbb{Z}_{L-1}$, using the Share Convert protocol `SC`.
3. P_0 and P_1 then compute the most significant bit of y by running `ComputeMSB(y)` and learns shares of the value: α.
4. Finally P_0 and P_1 outputs shares of α as $\gamma_j = j - \alpha_j + u_j$ where $j \in \{0, 1\}$

3.2.3 ReLU

The algorithm ReLU (`ReLU`) is realized by P_0 and P_1 holding shares of a over the ring $\mathbb{Z}_L$ and end with shares of `RELU(a)` over the ring $\mathbb{Z}_L$. Trivially it is seen that `RELU(a)` $= a$ if `MSB(a)` $= 0$, else 0. That is, RELU(a) = `RELU'(a)`$\cdot a$.

Input: P_0 and P_1 hold shares of a over the ring $\mathbb{Z}_L$
Output: P_0 and P_1 get shares of `ReLU(a)`

1. P_0 and P_1 gets shares of α by computing `RELU'(a)`.
2. P_0 and P_1 compute shares of c by calling `MatMul`, such that $c = \alpha \cdot a$.
3. Finally P_0 and P_1 outputs their shares of c masked by the common randomness u.

3.2.4 Division

Division might be considered a supporting protocol, however in *SecureNN* they implement `DIV` using the other supporting protocols, rather than from scratch. Therefore it is considered a main protocol both in this report and in *SecureNN*. Division (`DIV`) is a protocol that computes the division between two numbers `x` and `y` without any party knowing `x`, `y`, or x/y. P_0 and P_1 holds shares of `x` and `y` over $\mathbb{Z}_L$. At the end of the protocol P_0 and P_1 hold shares of x/y over $\mathbb{Z}_L$ when $y \neq 0$. The implementation uses long division, where the quotient is computed bit by bit sequentially, starting at the most significant bit

Input: P_0 and P_1 hold shares of `x` and `y`
Output: P_0 and P_1 gets shares of x/y.

1. P_0 and P_1 set a zero mask value $u_\lambda = 0$
2. **for** i $= \{\lambda - 1, ..., 0\}$ **execute** steps 3-7:
3. P_0 and P_1 each compute their share of z_i such that $z_i = x - u_{i+1} - 2^i y + w_{i,j}$ where $j \in \{0, 1\}$
4. P_0, P_1, and P_2 run `DReLU` with z_i as input and P_0 and P_1 learn each their share of β_i
5. P_0, P_1, P_2 call `MatMul` with inputs: β_i and $2^i y$. P_0 and P_1 learn their shares of v_i respectively.
6. P_0 and P_1 compute each their share of $k_i = 2^i \cdot \beta_i$
7. P_0 and P_1 compute each their share of u_i such that $u_i = u_{i+1} + v_i$
8. P_0 and P_1 computes each their share of the quotient $q = \sum_{i=0}^{l-1} k_i + s_j$ where s_j are shares of 0 for masking.

3.2.5 Maxpool

The Maxpool (`MP`) algorithm computes the maximum of n values. P_0 and P_1 hold shares of $\{x_i\}_{i \in n}$ over $\mathbb{Z}_L$ and at the end of the algorithm holds fresh shares of $\texttt{max}(\{x_i\}_{i \in n})$. The algorithm runs in $n - 1$ steps and recursively computes the max value of i and the previous value. So, $\texttt{max}_1 = x_1$, $\texttt{max}_2 = \texttt{max}(x_1, x_2)$, and finally $\texttt{max}_i = \texttt{max}(x_1, x_2, ..., x_i)$. The protocol is as such: first, compute shares of $w_i = x_i - \texttt{max}_{i-1}$ then compute shares of $\beta_i = \texttt{ReLU'}(w_i)$. If w_i is positive then $\texttt{ReLU'}(w_i)$ outputs 1, and if negative it outputs 0. Then call `SS` to select between x_i and $max_i - 1$, using β_i to compute $\texttt{max}_i$. Similarly we can compute the index of the maximum value, k such that $x_k = \texttt{max}(\{x_i\}_{i \in n})$, which is needed for prediction and the derivative of Maxpool in backpropogation.

Input: P_0 and P_1 hold shares of x_i, where i $\in n$
Output: P_0 and P_1 gets fresh shares of $\texttt{z}$ where $\texttt{z} = \texttt{max}(x_i)$ for $i \in n$.

1. P_0 and P_1 each sets their share of $\texttt{max}_1 = x_1$ and their share of $\texttt{ind}_1 = j$ where $j \in \{0, 1\}$
2. for $i = \{2, ..., n\}$ **execute** steps 3-7:
3. P_0 and P_1 compute each their share of $w_i = x_i - \texttt{max}_{i-1}$
4. P_0, P_1, and P_2 call `DReLU` with w_i as input, and P_0 and P_1 learn β_i
5. P_0, P_1, and P_2 call `SS` with β_i, $\texttt{max}_{i-1}$, and x_i as input and P_0 and P_1 learn $\texttt{max}_i$
6. P_0 and P_1 each set their share of $k_i = j \cdot i$ for $j \in \{0, 1\}$
7. P_0, P_1, and P_2 call `SS` with β_i, $\texttt{ind}_{i-1}$, and k_i as input and P_0 and P_1 learn $\texttt{ind}_i$
8. P_0 and P_1 outputs $\texttt{max}_n + u_j$ and $\texttt{ind}_n + v_j$, where u and v are shares of 0, so the outputs are masked.

3.2.6 Derivative of Maxpool

Derivative of Maxpool (`DMP`) is used for backpropogation and it is defined as a unitvector containing 1 in the index of the maximum value. *SecureNN* presents a specific implementation for the often-used case of 2x2 Maxpool. A more general case is described in Appendix D in *SecureNN*. The protocol exploits the fact that when `n` divides `L` we have a `mod` $n = (a$ `mod` $L)$ `mod` n.

Input: P_0 and P_1 holds each their share of x_i where $i \in |n|$
Output: P_0 and P_1 each get shares of z_i where $z_i = 1$ when $x_i =$`max(`$\{x_i\}_{i \in n}$`)` and 0 otherwise.

1. P_0, P_1, and P_2 call `MP` with input x_i and obtains shares of $\texttt{ind}_i$ from the second part of maxpools output
2. P_0 sends a share of $k = \texttt{ind}_n + r$ to P_2 where r is a random integer in $\mathbb{Z}_L$, while P_1 sends a share of $k = \texttt{ind}_n$ to P_2.
3. P_2 computes `t` such that $t =$ `Reconst(k)` using the two shares of k, and computes $k = t$ `mod` n and creates shares of E_k, and sends them to P_0 and P_1 respectively.
4. P_0 and P_1 each "left shifts" their respective shares of E_k by g such that $g = r$ `mod` n. Specifically when $E = (E_0, E_1, ..., E_{n-1})$, P_0 and P_1 computes each their share of D such that $D = (E_{-g \ mod \ n}, E_{1-g \ mod \ n}, ..., E_{n-1-g \ mod \ n})$
5. Finally P_0 and P_1 outputs their shares of $D + U$ where U are shares of 0 which masks the output.

3.2.7 End-to-end Protocols

All the described main protocols can be combined as needed in order to create a general end-to-end protocol used in the network. For example we can construct a small 3-layer neural network where the first layer is a fully connected layer, followed by a `ReLU` activation function, followed by another fully connected layer, followed by another `ReLU` activation, and finally a peseudo `softmax` output function $ASM(u_i) = \frac{ReLU(u_i)}{\sum ReLU(u_i)}$. The sequence of protocol calls which computes this network would look as such:

1. Call `MatMul` (1st layer weights)
2. Call `ReLU` (1st layer activation)
3. Call `MatMul` (2nd layer weights)
4. Call `ReLU` (2nd layer activation)
5. Call `DIV` (Softmax output ASM(u_i))

Backpropogation is computed by making calls to `MatMul` and `DReLU` with the appropriate dimensions. Similarly, a Convolutional Neural Network can be set up using a sequence of `MatMul` calls with `Maxpool` activations in between. These protocols can easily be combined and called sequentially since all protocols maintain the invariant that parties start with arithmetic shares of inputs and complete the protocol with arithmetic shares of outputs.

3.3 SecureNN Results

3.3.1 Communication and rounds

Figures 4 and 5 from *SecureNN* [1] shows the communication complexity of the supporting protocols and main protocols respectively.

Protocol	Rounds	Communication
MatMul$_{m,n,v}$	2	$2(2mn + 2nv + mv)\ell$
MatMul$_{m,n,v}$ (with PRF)	2	$(2mn + 2nv + mv)\ell$
SelectShare	2	5ℓ
PrivateCompare	1	$2\ell \log p$
ShareConvert	4	$4\ell \log p + 6\ell$
Compute MSB	5	$4\ell \log p + 13\ell$

Figure 4: Round & communication complexity of supporting protocols from *SecureNN* [1]

Protocol	Rounds	Communication
Linear$_{m,n,v}$	2	$(2mn + 2nv + mv)\ell$
Conv2d$_{m,i,f,o}$	2	$(2m^2f^2i + 2f^2oi + m^2o)\ell$
DReLU	8	$8\ell \log p + 19\ell$
ReLU (after DReLU)	2	5ℓ
NORM(l_D) or DIV(l_D)	$10l_D$	$(8\ell \log p + 24\ell)l_D$
Maxpool$_n$	$9(n-1)$	$(8\ell \log p + 29\ell)(n-1)$
DMP$_n$ (after Maxpool)	2	$2(n+1)\ell$

Figure 5: Round & communication complexity of main protocols from *SecureNN* [1]

`ReLU` and `DMP` requires calls to `DReLU` and `MP` respectively, so their total rounds would include these protocols. However for the sake of separation the rounds in `ReLU` and `DMP` are computed after the use of `DReLU` and `MP`. All communication is measured for λ-bit inputs and p is the field size 67. Complexities in the main protocols are presented using the optimized protocol of `MatMul` using PRF's[2] for correlated randomness. Compared to using garbled circuits[3], *SecureNN* claims to have larger than 8x improvement in communication complexity.

3.3.2 Secure training and inference

SecureNN evaluate their implementation using 4 different neural network architectures, in both training and inference on the MNIST dataset [16]:

- **Network-A** is a 3-layer Deep Neural Network (from [14])
- **Network-B** is a 4-layer Convolutional Neural Network (from [12] and [13])

[2] Pseudo Random Function
[3] [21]

- **Network-C** is a 4-layer Convolutional Neural Network (from [11])
- **Network-D** is a 3-layer Convolutional network (from [13] and [15])

For training they evaluate their protocols for network **A**, **B**, and **C** in both a LAN and WAN setting where they achieve $\sim$ 99% accuracy on their test datasets. They vary between 5-15 epochs[4] on all the networks except **A**, since this network does not achieve good accuracy for smaller epochs. Furthermore they vary their batch size[5] between 4 and 128 for networks **B** and **C**. Their results are presented in figure 6 and 7, which are tables from *SecureNN*.

	Epochs	Accuracy	LAN (hours)	WAN (hours)
A	15	93.4%	1.03	7.83
B	5	97.94%	5.8	17.99
	10	98.05%	11.6	35.99
	15	98.77%	17.4	53.98
C	5	98.15%	9.98	30.66
	10	98.43%	19.96	61.33
	15	99.15%	29.95	91.99

Figure 6: Secure training execution times for batch size 128

	Batch size	Accuracy	LAN (hours)	WAN (hours)
B	4	99.15%	9.98	112.71
	16	98.99%	8.34	36.46
	128	97.94%	5.8	17.99
C	4	99.01%	18.31	123.96
	16	99.1%	13.43	46.2
	128	98.15%	9.98	30.66

Figure 7: Secure training execution times for 5 epochs

Their results shows that there is a clear trade-off between accuracy and speed, which highly depend on the hyper parameters epoch and batch size. Generally they achieve higher accuracy by increasing the epoch count, as it allows the `Gradient Decent` optimization [17] (in back-propagation) to converge to an optimum, but at the cost of speed. Similarly by reducing batch size they can increase accuracy further, but again at the cost of speed. This is a standard machine learning problem, where it is in the hand of the programmer to find a desirable sweet spot between accuracy and speed. Depending on the domain it may be more valuable to achieve higher accuracy, and not care so much about the training time, since the training process only needs to happen once, or a few times (in case they want to update their model at a later point).

[4]One epoch: one forward and backwards pass for entire dataset
[5]batch size: amount of training examples used for a single forward and backwards pass

SecureNN also evaluate their protocols for inference (prediction) on all 4 networks: **A**, **B**, **C**, and **D**. These results are presented in figure 8, from *SecureNN*.

	LAN (s)		WAN (s)		Comm (MB)	
Batch size →	1	128	1	128	1	128
A	0.043	0.38	2.43	2.79	2.1	29
B	0.13	7.18	3.93	21.99	8.86	1066
C	0.23	10.82	4.08	30.45	18.94	1550
D	0.076	2.6	3.06	8.04	4.05	317.7

Figure 8: Prediction timings for batch size 1 vs 128 for *SecureNN* over MNIST

Obviously it shows that WAN is slower than LAN, and inferring 128 inputs is slower than inferring 1 input. However we can see that all four networks can infer a single input over LAN in less than 1 second, and over WAN in $\sim$ 2-4 seconds. Network **B** and **C** seems to scale a lot worse than **A** and **D** when increasing to 128 inputs. However **B** and **C** also achieved significantly higher accuracy than **A**, so this might be negligible. It's interesting however that **D**, seems to scale a lot better in terms of input size, than **B** and **D**, considering they were able to achieve 99%[6] accuracy on **D**.

3.4 Communication bottleneck

SecureNN is able to achieve good results in terms of accuracy on the data sets they test on. Whether they are able to achieve consistently high accuracies for other data sets is left for another paper to explore. We will therefore, in this paper, not explore how *SecureNN* can be improved on the machine learning side, but rather how their protocols can be optimized in terms of speed.

Analyzing the communication tables 4 and 5 from *SecureNN* it shows that `Share Convert` and `ComputeMSB` uses a lot of rounds of communication (4 and 5 respectively), and these protocols are used extensively. Since `ComputeMSB` is effectively used to determine the sign of values for `ReLU` and `DReLU`, and each call to `ComputeMSB` requires a prior call to `Share Convert`, it's clear that this can easily become a bottleneck for communication. Therefore it's highly desirable to reduce the amount of rounds required as well as data transferred when computing the MSB. In the following sections we will discuss alternative approaches for computing the MSB and present our experiments comparing different protocols.

[6]This is written in their appendix, but is not depicted in their tables of training execution times

4 Reducing communication in SecureNN

4.1 Initial attempt by removing Share Convert

Based on our research project conducted in Autumn 2019, we had an initial hypothesis we wanted to test: Removing `SC` (Share Convert) entirely and running `ComputeMSB` (hence assuming all values are already in $\mathbb{Z}_{L-1}$). This idea was based on the assumption that `ComputeMSB` functions properly as long as shares are in $\mathbb{Z}_{L-1}$ and thus `SC` only handles edge cases where shares are exactly $\mathbb{Z}_{L-1}$. Under this assumption, `ComputeMSB` only produces wrong results if input shares are exactly $\mathbb{Z}_{L-1}$, therefore extremely rarely. Our hypothesis stated that this introduces an insignificant amount of noise to the neural network which can be ignored given the robustness of neural networks towards relatively small noise, but greatly increasing the overall communication.

In an attempt to test this hypothesis by manually removing Share Convert from the source code of *SecureNN* and running the same tests they did, we ran into several complications. First of all, we spent a fair amount of time attempting to compile and run their code. However, we were never able to reproduce their original results (or even come close). Secondly, it seemed to contain bugs in its implementation of various protocols (including `ComputeMSB`). So instead we decided to implement `Share Convert` and `ComputeMSB` ourselves and perform experiments on these. However, during this process we found out that our hypothesis was inherently flawed. The assumption that `ComputeMSB` works as long as shares are in $\mathbb{Z}_{L-1}$, was wrong. In fact, it works if shares are in $\mathbb{Z}_{L-1}$ but also only if the reconstructed (summed) value of the shares are in $\mathbb{Z}_{L-1}$. This, means that `ComputeMSB` will produce wrong results when both shares are larger or equal than half of $\mathbb{Z}_L$, which happens $\sim 25\%$ of the time, which is far too often.

So it was not an option to compute the MSB without first converting shares to $\mathbb{Z}_{L-1}$. Therefore we changed our focus and started looking into alternative approaches to MSB computation.

4.2 Alternative approach to private machine learning

During this project, *'High Performance Logistic Regression for Privacy-Preserving Genome Analysis' by De Cock et. al* [2] (*The Genome Paper*) was published which claims to achieve fast and secure Logistic Regression by using a two party protocol together with a trusted initializer (for distribution of correlated randomness). They propose a new efficient method for computing activation functions which doesn't require secure comparison (unlike *SecureNN*) nor Yao's garbled circuits[7]. Instead they make use of a highly optimized bit-decomposition protocol in order to realize their activation function protocol. Their implementation won first prize in Track 4 of the iDash 2019 secure genome analysis competition. While their paper only presents a protocol for secure Logistic Regression, it's easy to see how some of the methods they use can be extended to

[7] [21]

apply to neural networks.

In terms of our focus, it's interesting to look at how they perform fast bit-decomposition, since this can be used as an alternative to computing the MSB. *The Genome Paper* presents a naive linear protocol for bit-decomposition and a highly optimized logarithmic version. Using an efficient bit-decomposition protocol to compute the MSB instead of doing `SC` and `ComputeMSB` (which inherently also uses `PC`), might turn out to be a good alternative which reduces communication and ultimately speeds up *SecureNN*.

Therefore, for the remainder of this report we describe the secure bit-decomposition protocols from *The Genome Paper*, and present our own experiments in which we compare and analyze the two approaches;
optimized bit-decomposition (`BitDecompOPT`) vs `SC` and `ComputeMSB`.

4.3 Bit-decomposition

4.3.1 Naive bit-decomposition

An initial naive method for doing secure 2-party bit-decomposition is described in *The Genome Paper* by means of ripple carry addition on an adder circuit, which has linear communication complexity (`BitDecomp`). When we want to calculate if there is a carry-bit in the i'th position, namely c_i, we see that there are two cases where this is true. In the first case we have that both $a = 1$ and $b = 1$ which is equivalent to computing $a \cdot b$. In the second case we have a carry-bit from c_{i-1} and also $a + b = 1$.

We can therefore compute $d_i = a_i \cdot b_i + 1$ which corresponds to the first case. Then compute $e_i = (a_i + b_i) \cdot c_{i-1} + 1$ which corresponds to the second case. We observe the addition of 1 in the end of both cases. The idea is to *flip* the statement, meaning that d_i and c_i will both be logically equal to their respective cases, where there is no carry-bit.

`if` $d_i == 1$ `then the first case does not produce a carry bit`
`if` $e_i == 1$ `then the second case does not produce a carry bit .`

Ultimately to find out if there should be a carry bit at the i'th position ($c_i = 1$) we compute the following: $c_i = d_i \cdot e_i + 1$.
Now only if both e_i and d_i are 1 then there will be no carry bit, otherwise there will be a carry bit.

Input: P_0 and P_1 hold shares of a secret S in $\mathbb{Z}_L$.
Output: P_0 and P_1 gets XOR shares of the bit-decomposition of S.
Common Randomness: P_0 and P_1 hold shares of beaver triplets for one round matrix multiplication. These have been pre-distributed by the trusted initializer.
Preface: All multiplications are over $\mathbb{Z}_2$. In a bitstring, $i = 1$ denotes the least significant bit, $i = \lambda$ denotes the most significant bit.

1. P_0 and P_1 each addresses their respective shares as the corresponding bit

string (a_i and b_i). Now we denote the shares of y_i as the reconstruction of the pair of shares (a_i,b_i) where $y_i = a_i + b_i \ mod\ 2$. Now define the shares of a_i as: (a_i,0) and b_i as: (0,b_i).

2. Compute $c_i = a_1 \cdot b_1$ and $x_1 = a_1 + b_1$.
3. **For** $i = 2$ to λ **execute** steps 4-7:
4. $d_i = a_i \cdot b_i + 1$
5. $e_i = (a_i + b_i) \cdot c_{i-1} + 1$
6. $c_i = d_i \cdot e_i + 1$
7. $x_i = a_i + b_i + c_i$
8. Finally output $x_i ... x_\lambda$ as valid XOR shares of the bit-decomposition of the secret S

4.3.2 Optimized logarithmic bit-decomposition

The highly optimized bit-decomposition protocol (`BitDecompOPT`) can be realized in logarithmic communication complexity and is based on an approach called *speculative* adder circuit in which at each layer the next layers carry bits are computed speculatively, once for each case that the previous carry bit had been 0 and 1. This increases local computation, but reduces rounds of communication to $\log(\lambda) + 2$ and total data transfer to $4\lambda \cdot log(\lambda) + 6\lambda$ bits. *The Genome Paper* proposes an even more optimized version further reducing the number of communication rounds by 1 (or 2, in special cases), and requires a small fraction of the data transfer cost of the speculative adder circuit.

When summing the binary numbers a and b, we have the i-th bit represented as $s_i = a_i \oplus b_i \oplus c_{i-1}$ (s_i corresponds to x_i in the naive protocol) and $c_i = a_i b_i \oplus a_i c_{i-1} \oplus b_i c_{i-1}$. Alternatively this means that the carry depends on two signals which each depends on a and b. *Generate* ($g_i = a_i b_i$), creates a new carry bit at the i-th position, and *Propogate* ($p_i = a_i \oplus b_i$) accounts for the previous carry bit if it exists. In this view, we have $s_i = p_i \oplus c_{i-1}$ and $c_i = g_i + p_i c_{i-1}$. This is a sum of products which can be expressed as the following matrix multiplication:

$$\begin{bmatrix} c_i \\ 1 \end{bmatrix} = \begin{bmatrix} p_i & g_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{i-1} \\ 1 \end{bmatrix} = M_i \begin{bmatrix} c_{i-1} \\ 1 \end{bmatrix}$$

When multiplying two matrices of the form M_i, the lower entries don't change; They remain as 0 and 1 respectively:

$$\begin{bmatrix} c_i \\ 1 \end{bmatrix} = \begin{bmatrix} p_i & g_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_{i-1} & g_{i-1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{i-2} \\ 1 \end{bmatrix} = M_i M_{i-1} \begin{bmatrix} c_{i-2} \\ 1 \end{bmatrix}$$

Utilizing this, we observe that it is possible to compute the carry c_i at all bit positions, by computing the set of all matrix compositions:

$$\prod_{j=1}^{i} M_j \quad | \quad 1 \leq i \leq \lambda$$

Since we know that the carry-in to the first bit can be represented as the vector $\binom{0}{1}$ for all j, we can derive c_i implicitly as the upper right hand entry of $M_{1.i}$ ($M_{1.i}$ denoting the resulting matrix composed of all matrices M_1 through M_i consecutively) [2].

Therefore, in order to efficiently compute the set of all matrix compositions $M_{1.i}$ *The Genome Paper* proposes the construction of a matrix composition network (`ComposeNet`) with logarithmic depth, where at the i-th layer all compositions $M_{1.j}$ that require fewer than 2^{i-1} compositions are computed. The network is constructed such that each $M_{1.j}$ is the composition of the "largest" matrix (contains most compositions) from the previous layer. For $j = \{1, ..., \lambda - 1\}$ $M_{1,j}$ are inserted with the parents having the values $M_{1.2^{i-2}}$, and $M_{2^{i-2}+1.j}$, where i denotes the current layer. If the parents don't exist in the network, they are added recursively. Figure 9 shows an example with $\lambda = 17$ taken from *The Genome Paper*.

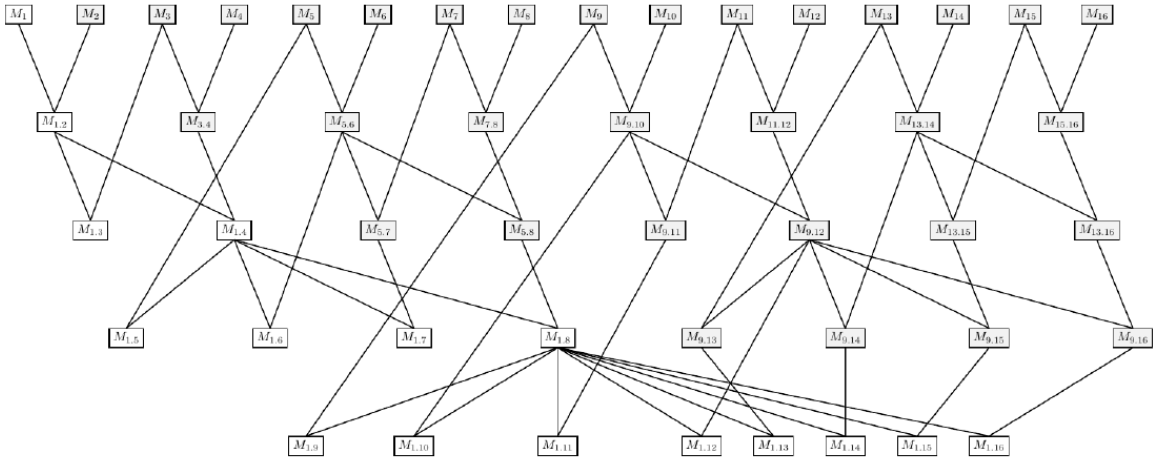

Figure 9: Composenet$_\lambda$ for $\lambda = 17$. Computes the set of all matrix compositions $M_1, M_{1.2}, M_{1.3}, ..., M_{1.(\lambda-1)}$

Input: P_0 and P_1 hold shares of a secret S in $\mathbb{Z}_L$.
Output: P_0 and P_1 gets XOR shares of the bit-decomposition of S.

1. P_0 and P_1 regards its shares of S as bit-strings $p_{0,1}, ..., p_{0,\lambda}$ and $p_{1,1}, ..., p_{1,\lambda}$ s.t.

2. $p_j = p_{0,j} \oplus p_{1,j}$ for j = 1,...,λ

3. P_0 creates the sharing $g_{0,j} = (p_{0,j}, 0)$
4. P_1 creates the sharing $g_{1,j} = (0, p_{1,j})$
5. $g_j \leftarrow g_{1,j} \cdot g_{2,j}$
6. $M_j \leftarrow \begin{bmatrix} p_j & g_j \\ 0 & 1 \end{bmatrix}$ for all j
7. $\{M_{1.j} \mid 1 \leq j < \lambda\} \leftarrow \texttt{ComposeNet}_\lambda(\mathrm{M})$. Obtain the set of matrix compositions $M_{1.j}$ from `ComposeNet`
8. $c_j \leftarrow$ The upper right entry of $M_{1.j}$
9. $s_1 \leftarrow p_1$. The least significant bit has no carry-in so it's equal to the LSBs of the two bit shares
10. $s_j \leftarrow p_j \oplus c_{j-1}$ for all $j > 1$
11. **return** s_1,...,s_λ

5 Experiments

In order to analyze and compare *SecureNNs* approach (using `SC` and `ComputeMSB`), with the optimized bit-decomposition approach (`BitDecompOPT`) we implemented the following protocols in Python 3.7:

- Multiplication 2-party: `Single-MatMul`, `Single-Mult`, `List-MatMul`, `List-Mult`
- Multiplication 3-party: `Single-Mult`
- `SC` (Share Convert)
- `PC` (Private Compare)
- `ComputeMSB`
- `ComposeNET`
- `BitDecomp`
- `BitDecompOPT`

Naturally we have also implemented the underlying architecture for supporting arithmetic on big integers, networking, secret sharing and everything else needed to properly simulate the protocol. Despite Python being a relatively slow language, it should suffice since all protocols are implemented in this language and thus share the same overhead. The experiments have been run locally and distributed. The tests have been conducted locally on windows and distributed on linux, it is worth noting that there exists many intricate differences regarding networking for both operating systems, but we will ignore that discussion in this paper.

Our source code is available on github:
https://github.com/Aramos93/ComputeMSB_BitDecompOPT_Experiments
(http://tiny.cc/computemsb)

The main branch contains the local Windows version and there exists two branches for `BitDecompOPT` and `ComputeMSB` respectively, for linux.

5.1 Our ComposeNet implementation

5.1.1 Generating ComposeNets for different bit length integers

The purpose of `ComposeNet` is to construct a logarithmic depth network corresponding to the bit-length of the value that is being bit-decomposed. A given `ComposeNet` is a minimum tree representing the required matrix multiplications needed to compute shares of all individual bit values. The tree consists of layers, where each layer references the previous and is created from composing matrices in the previous layers. Each node in the network represents a matrix composition e.g. $M_{1.2}$ which is the matrix composed of M_1 and M_2, and similarly $M_{1.3}$ is composed of $M_{1.2}$ and M_3. Each node holds such two values, indicating which matrices it's composed of, as well as a reference to those two matrices.

Input: An integer λ representing the bit-length of the input integer.
Output: A graph like object representing the `ComposeNet`. The graph is represented as a list, where each element of the list is another list containing the required compositions to achieve a given matrix composition.

1. let number of layers, M be $M = \lceil log2(\lambda - 1) \rceil + 1$
2. Define a function `insert(`$M_{X.Y}$`)`, such that `insert(`$M_{X.Y}$`)`:
   `let i =` $M_{X.Y}$`.layer`
   `if` $M_{X.X+2^{i-2}-1}$ `NOT EXISTS; THEN insert(`$M_{X.X+2^{i-2}-1}$`)`
   `if` $M_{X+2^{i-2}.Y}$ `NOT EXISTS; THEN insert(`$M_{X+2^{i-2}.Y}$`)`
3. **For** i = 1 to $\lambda - 1$ **execute** step 4:
4. Construct $M_{i.i}$ in the first layer
5. **For** i = 2 to $\lambda - 1$ **execute** step 6:
6. Call `Insert`$(M_{1.i})$

5.1.2 Reducing communication by wrapping matrices in bytes

Instead of sending serialized pickled[8] python objects over the network, we wrap the contents of the matrices in single bytes to reduce communication. The content of the matrices are all modulo 2, which means their values can be represented by bits in a byte. We iterate over the 2x2 matrices and extract the 4 bit values, which are then appended to 4 zeroes, and then stored in a byte, which is then stored in a byte array. This reduces the communication to about a 13-th of sending the pickled objects. Since bytes are 8 bits, we could have achieved even less communication by storing the values of 2 matrices in one byte, but due to time constraints and implementation complexity this wasn't done. An overview can be seen in figure 10.

Another improvement which could have been implemented was to encode 4 matrices in 1 byte, this could have been achieved since the *lower* two values of the matrices never change, and thus do not need to be sent over the network.

[8] A python library handling object serialization. https://docs.python.org/3/library/pickle.html

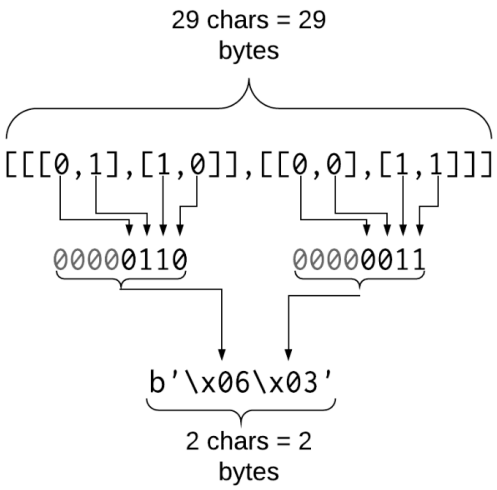

Figure 10: A visual representation of the wrapping from lists of matrices to bytes

5.2 Design of experiments

In order to properly measure the running time of both protocols, we run each protocol 1000 times using the same inputs. We measure wall clock time and amount of data transferred (in bytes). Just like *SecureNN* we use $L = 2^{64}$ and $p = 67$. For both protocols we measure total time to finish 1000 inputs as well as total bytes sent. We use these numbers to compute the average time and bytes per call to `BitDecompOPT` and `SC` + `ComputeMSB`. Furthermore we measure the average time and bytes sent per call to `SC`, `ComputeMSB`, and `PC`. In `BitdecompOPT` we don't measure time for any subroutines. These timings allow us to get an indicative average of both protocols, as well as individual sub-protocols.

5.2.1 Local experiements

In the local setting we tested the experiment on a Windows 10 machine with 4 cores and 16 gigabytes of ram. The experiment was tested with web sockets connecting to local ports as to simulate networking, it is worth noting that local networking still has a roundtrip overhead, however very small, a local ping takes 0.19 ms per roundtrip.

5.2.2 Distributed experiements

In order to properly asses the overhead of communication, we decided to run the experiment in a distributed setting using servers from DigitalOcean.com. We set up 3 ubuntu 18.04 lts (64 bit) machines, each with 1 core and 1 gigabyte of ram. The machines were located in London, Amsterdam and Frankfurt. The roundtrip times obtained by ICMP pinging between the machines were:

- London-Frankfurt: 12.23 ms

- London-Amsterdam: 8.05 ms
- Amsterdam-Frankfurt: 18.21 ms

To properly compare a 2 party protocol with a 3 party protocol we would need distributed machines with equal ping between them, which we couldn't obtain with our resources. Instead, to fairly compare the two protocols we compare the running time of the 3 party protocol with the average running time between each pair of servers for the 2 party protocol.

6 Results

In table 1 the the first three results named `BitDecomptOPT` **A**,**B**, **C** are the pairwise timings between the three distinct servers; **A**: London-Amsterdam, **B**: London-Frankfurt, **C**: Amsterdam-Frankfurt. `BitDecompOPT` **Avg** shows an average of these. The interesting comparison is between `BitDecompOPT` **Avg** and `SC + ComputeMSB`. The results show that `SC + ComputeMSB` is faster than `BitDecompOPT` when run locally. However once we move to a distributed setting `BitDecompOPT` becomes ∼25% faster. In terms of communication, `SC + ComputeMSB` sends slightly more data during its protocol.

	Local time (s)		**Dist. time (s)**		**Comm (bytes)**	
Input size →	**1**	**1000**	**1**	**1000**	**1**	**1000**
`BitDecompOPT` **A**	-	-	0.0686	68.58	1674	1674000
`BitDecompOPT` **B**	-	-	0.0726	72.58	1674	1674000
`BitDecompOPT` **C**	-	-	0.0825	82.47	1674	1674000
`BitDecompOPT` **Avg**	0.0304	30.48	0.0745	74.54	1674	1674000
`SC + ComputeMSB`	0.0221	22.04	0.1015	101.52	1987.493	1987493

Table 1: local vs distributed times and data transfer for 1 and 1000 inputs

6.1 Subroutine timings

Table 2 shows times and data transfer of the subroutines within `SC + ComputeMSB`. We can see that `SC` and `ComputeMSB` takes the most time while `PC` is actually pretty fast (presumably because the protocol has very little communication overhead). Similarly `SC` and `ComputeMSB` each transfers a significant amount of data

	Local time (s)		**Dist. time (s)**		**Comm (bytes)**	
Input size →	**1**	**1000**	**1**	**1000**	**1**	**1000**
`PC`	0.00497	4.97	-	-	330.22	330220
`SC`	0.00979	9.79	0.05115	51.15	851.49	851490
`ComputeMSB`	0.01225	12.25	0.04886	48.86	1136	1136000

Table 2: Individual times of subroutines in ShareConvert and ComputeMSB for 1 and 1000 inputs

6.2 Raw local computations

Table 3 shows time spent purely performing local computations. The result clearly shows that `BitDecompOPT` spends a considerable amount of time on local computation compared to `SC` + `ComputeMSB`; roughly 75.5x more.

	time (s)	
Input size →	**1**	**1000**
`SC` + `ComputeMSB`	0.000147	0.147
`BitDecompOPT`	0.0111	11.1

Table 3: Raw local computation times for 1 and 1000 inputs (all communication time excluded)

Results are further discussed in the following section 7

7 Discussion

7.1 ComputeMSB versus BitDecompOPT in local setting

As can be seen from the results, `ComputeMSB` runs faster in the local setting than `BitDecompOPT`, which initially may seem surprising since `BitDecompOPT` performs fewer communication rounds. However, there may be several reasons for this. One explanation may be that our implementation of `BitDecompOPT` is not sufficiently optimized. Our implementation uses sockets to communicate between processes, which causes a big overhead as opposed to using globally accessible variables for communication or unix sockets, which allow for communication between processes without having the overhead of moving up and down the network stack. Another explanation lies in the trade-off that `BitDecompOPT` does in order to reduce rounds. Namely, we allow `BitDecompOPT` to perform far more local computation in order to reduce rounds and data transferred. This works under the assumption that our hardware is good enough to perform these extra computations where as distributed communication is not as reliable in terms of speed, therefore we willingly sacrifice local computation to save rounds. In a local setting, there is barely any latency hence we don't get much benefit from the `BitDecompOPT` protocol.

7.2 ComputeMSB versus BitDecompOPT in distributed setting

When latency is induced artificially (with sleeps in the protocol) or naturally (by running the protocol distributed) we see that `BitDecompOPT` runs faster than `ComputeMSB`, which is in line with our hypothesis. We observe that even though our python implementation has larger computational and memory overhead than the *Rust* implementation used in *The Genome Paper*, the speedup still exists through round reduction. We also observe that even though `BitDecompOPT` has far larger local execution time than `ComputeMSB`, the time a round takes is greater than the extra local execution time. This supports the idea that reducing rounds is crucial in MPC, especially when servers are cross country and even more so if cross continental.

7.3 Theoretical communication versus actual communication

In both *SecureNN* and *The Genome Paper*, when discussing the amount of data transfer, the number of bits sent is a theoretical minimum amount. When working with these protocols in higher level languages it is virtually impossible, or at least extremely complex doing bit by bit computation and communication. In our implementation we use python3 integers which, if not exceeding size, is held in 64 bit addresses (assuming a 64 bit architecture), this causes a big overhead compared to the theoretical values. In these papers, researchers spend significant time implementing protocols in low level languages and implement

them to run highly parallelized in order to squeeze as many individual bits to be computed on, into 32 or 64 bit addresses. For us to achieve similar results, we would need considerably more time spent on implementation. This can be the reason that we sometimes see quite larger amounts of data transferred and quite high computation times.

7.4 Reusing beaver triplets in BitdecompOPT for less data transfer

The matrices at each node at each layer of `ComposeNET` are reused extensively when constructing the `ComposeNET`, an example; we see that $M_{9.10}$ in figure 9 are used twice in the third layer and once in the fifth layer. In our implementation the nodes' matrix values are masked by distinct beaver triplets. In *The Genome Paper* they re-use correlated randomness where information leakage is not possible to reduce communication, this in essence means that the node $M_{9.10}$ is only transferred once, even though its used multiple times. We have not implemented this redundancy check, we simply let the transfer happen multiple times.

7.5 Precomputing common randomness opposed to a trusted initializer

Having a trusted initializer is a strong trust assumption. It would be better if we could remove that assumption and replace it with a protocol that does the same but without relying on that trust. There exists protocols that achieve this, but have not been utilized in neither *SecureNN* or *The Genome Paper*. We see protocols like these in *Multiparty Computation from Somewhat Homomorphic Encryption*[19] and in *Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits* [20]. Implementing these protocols to run as part of an *offline* phase before running the actual protocol would remove the need for a trusted initializer. The protocol for preprocessing triplets have been proven UC secure and should only provide *SecureNN* or *The Genome Paper* less harder assumptions. The generation of these triplets should be done whenever the party is idle and not computing anything, since there is no online part to it.

7.6 Combining approaches to optimize SecureNN

`BitDecompOPT` has shown to be favourable to `ComputeMSB`, due to fewer rounds and also due to the fact that `BitDecompOPT` provides a full integer bit-decomposition, as opposed to only computing the MSB. Replacing the `SC` + `ComputeMSB` approach with `BitDecompOPT` in *SecureNN* may significantly reduce the training time of neural networks in a distributed setting, however this would require further testing. Both *The Genome Paper* and *SecureNN* assume a trusted initializer, they also both claim their protocols to be UC and they claim they've proven them UC secure, therefore it should be possible to move protocols between implementations without breaking any security guarantee. However porting protocols between *SecureNN* and *The Genome Paper* may require some

restructuring since *SecureNN* is based on a 3-party MPC structure, while *The Genome Paper* only uses 2 parties.

7.7 Utilizing the full bit-decomposition

SecureNN only computes the MSB of some value a which is used to compute `DReLU(a)` and subsequently `ReLU(a)`, where as *The Genome Paper* computes the entire (or partial) bit-decomposition of a in order to achieve the same thing; the MSB and in turn the ReLU activation function. Since both protocols are only interested in the MSB, the bit-decomposition approach ends up with extraneous information, namely the bit values of $x_{\lambda-1}, .., x_0$. However, *The Genome Paper* still chose this approach because there exists very fast and optimized protocols for performing bit-decomposition (e.g. `BitDecompOPT` as proposed by *The Genome Paper* itself), which ultimately spends less rounds than computing the MSB alone (which is constrained by the fact that shares needs to be converted to $\mathbb{Z}_{L-1}$ prior to the protocol). It seems wasteful to compute the bit-decomposition simply to obtain the MSB even though it's currently the fastest approach. However if the remaining bit-values from the result of the bit-decomposition could be utilized for another purpose, then performing bit-decomposition would become even more efficient.

The protocols as they currently are in *SecureNN*, does not benefit from knowing the full bit-decomposition of some value a. However, the protocol for performing long divison `DIV`, uses an approach which iteratively computes the resulting quotient bit by bit. It may be plausible to implement a faster version of `DIV` which utilizes all known bit-values obtained by a bit-decomposition. Secondly, both *SecureNN* and *The Genome Paper* only implements the `ReLU` activation function (because of its simplicity), but it may become desirable to invent protocols for other activation functions such as *Sigmoid*. *Gaussian*, *TanH*, etc. These functions are more complicated than ReLU and so one would imagine their corresponding MPC version would also be. Being able to efficiently compute the full bit-decompositions may prove to be helpful here, but all this is left for another paper to explore.

7.8 Future prospects for private machine learning

Private machine learning has been proven both theoretically and practically possible and there are various approaches being explored as in how to realize these protocols. However, there are still many optimizations to be made, both in terms of security and efficiency but also on the machine learning side. In this paper we have only investigated the efficiency of PPML protocols, but it would also be interesting to look at how protocols can be improved from a machine learning perspective. One major point of interest is suggested in *The Genome Paper*, which encourages the development of efficient and secure protocols to perform dimensionality reduction. Dimensionality reduction in machine learning is a powerful tool that can provide major speed-up to ML algorithms by selecting and using only the most important features from a data-set [24]. Concretely for our example of hospital records, this means that while there may be many features explaining the data (blood type, bodyfat, heart rate, number of

cell transformation, etc.) a lot of these features explain the same or no variance, effectively making them redundant. It's therefore desirable to select the most important features and only use these in the ML algorithm, ultimately providing a huge speed up, at a small cost of accuracy.

Specifically in *The Genome Paper*, where they perform logistic regression on genomes, they report an accuracy of 99.58% and 2.52 seconds training time when training on all 17814 features in a genome data-set. However by selecting a certain 54 features (part of a 76-gene signature set described in [25]), and training on these they still achieve 98.93% accuracy and 0.51 seconds run time - so about a 2 second runtime decrease, at the cost of 0.65% accuracy, achieved by removing 99.7% of all features. On a second data-set containing 12634 features, they had a runtime of 26.90 seconds and 64.82% accuracy. Unfortunately they weren't able to isolate the same 54 genes in this data-set (due to labeling issues in the data-set) and thereby test accuracy on the reduced number of features, but instead they chose 76 random genes to test the speed reduction and achieved a runtime of 6.71 seconds which is about a 20 second training time decrease.

This shows that if dimensionality reduction can be performed, run times can be significantly improved while still maintaining sufficient accuracy. There exists many intricate ML methods for performing dimensionality reduction but to our knowledge no solutions for secure dimensionality reduction in an MPC setting. "*Secure dimensionality reduction fusion estimation against eavesdroppers in cyber–physical systems*" by Daxing Xu et. al. [26] proposes a method where they introduce random noise in order to mask data, however it's desirable to invent a protocol which uses an MPC approach, which can be combined with the methods discussed in this paper.

8 Conclusion

In this thesis we have presented recent work within the field of privacy preserving machine learning, specifically methods using multi party computation with secret shares in order to construct secure neural networks. We described in detail the protocols proposed in [1] which uses 3 parties and we identified a communication bottleneck in how they compute the most significant bit. The MSB is computed very often in order to realize the ReLU activation function (as well as other functions) within neural networks. Reducing communication between parties in MPC is highly desirable, due to the inherent communication overhead. We describe an alternate solution using bit-decomposition proposed by [2] in order to obtain the MSB. This solution reduces the total amount of communication rounds and data transferred required, at the cost of more local computation.

Furthermore we presented our own implementation of the described protocols and subprotocols in Python 3.7 as well as our experiments comparing the two approaches in a local and distributed setting. In conclusion the results show that using the optimized bit-decomposition approach is faster and uses less communication in a distributed setting than computing the MSB directly. In a local setting `ComputeMSB` is faster as assumed, since there is barely any communication overhead.

Finally, we discuss our implementation, potential improvements and other factors that could impact the results. In terms of future work on the topic of PPML, there is room for improvement on both the security and the machine learning side, as well as general optimization of efficiency. For instance, even though bit-decomposition is the most efficient solution in a distributed setting, it still computes extraneous information which could potentially be utilized elsewhere in the protocol instead of going to waste, hereby improving the efficiency of bit-decomposition further. Additionally in order to thoroughly explore the machine learning side of the topic, there's a need for the development of secure protocols for dimensionality reduction, a variety of activation functions (*Sigmoid*, *Gaussian*, *TanH*), and various other machine learning models; e.g. support vector machines, decision trees, clustering algorithms. Ultimately our thesis provides an in depth description of how private machine learning can be realized by using MPC methods proposed in [1] to construct secure neural networks and further optimized with methods from [2].

Bibliography

[1] Wagh, Sameer and Gupta, Divya and Chandran, Nishanth. SecureNN: 3-Party Secure Computation for Neural Network Training. https://eprint.iacr.org/2018/442.pdf, 2019.

[2] Martine De Cock, Rafael Dowsley, Anderson C. A. Nascimento, Davis Railsback, Jianwei Shen, Ariel Todoki. High Performance Logistic Regression for Privacy-Preserving Genome Analysis. https://arxiv.org/abs/2002.05377

[3] Kumar, Rathee, Chandran, Gupta, Rastogi, and Sharma . CryptFlow: "*Secure TensorFlow Inference*. https://eprint.iacr.org/2019/1049.pdf, 2019

[4] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. "*High-throughput semi-honest secure three-party computation with an honest majority*". In ACM CCS, 2016.

[5] R. Canetti. Universally composable security: A newparadigm for cryptographic protocols. *In Proceedings of the 42Nd IEEE Symposium on Foundations of ComputerScience*, FOCS '01, pages 136–, 2001.

[6] Ran Canetti. Security and composition of multiparty cryp-tographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[7] Bishop, Christopher M. (2006). "*Pattern Recognition and Machine Learning*". Springer.

[8] Bengio, Yoshua; LeCun, Yann; Hinton, Geoffrey (2015). "*Deep Learning*". Nature. 521 (7553): 436–444.

[9] Donald Beaver. "*Efficient multiparty protocols using circuit randomization*". In Annual International Cryptology Conference, pages 420–432. Springer, 1991.

[10] Stanford CS231n: "*Convolutional Neural Networks for Visual Recognition*". https://cs231n.github.io/convolutional-networks/

[11] 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.

[12] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "*Oblivious neural network predictions via MiniONN transformations*". In ACM CCS, 2017.

[13] Payman Mohassel and Peter Rindal. ABY3: "*A mixed protocol framework for machine learning*". In ACM CCS, 2018.

[14] Payman Mohassel and Yupeng Zhang. SecureML: "*A system for scalable privacy-preserving machine learning*". In ieeeoakland, 2017.

[15] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko,Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: "*A hybrid secure computation framework for machine learning applications*". In AsiaCCS, 2018.

[16] MNIST database. http://yann.lecun.com/exdb/mnist/ Accessed: 05-12-2019.

[17] Claude Lemaréchal. "*Cauchy and the Gradient Method*" https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf

[18] Tord Ingolf Reistad, Tomas Toft: "*Linear, Constant-Rounds Bit-Decomposition*". ICISC 2009: 245-257

[19] Ivan Damgård, Valerio Pastro, Nigel P. Smart, Sarah Zakarias: "*Multiparty Computation from Somewhat Homomorphic Encryption*". CRYPTO 2012: 643-662

[20] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, Nigel P. Smart:" *Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits*". ESORICS 2013: 1-18

[21] A. C.-C. Yao. "*How to generate and exchange secrets*". In Foundations of Computer Science, 1986., 27th Annual Symposium on, pages 162–167. IEEE, 1986.

[22] Peter Bright - Jul 10, 2018 9:00 pm UTC (2018-07-10). *New Spectre-like attack uses speculative execution to overflow buffers.* Ars Technica. Retrieved 2018-11-02.

[23] Schwarz, Michael; Weiser, Samuel; Gruss, Daniel (2019-02-08). *Practical Enclave Malware with Intel SGX.* arXiv:1902.03256

[24] Roweis, S. T.; Saul, L. K. (2000). "*Nonlinear Dimensionality Reduction by Locally Linear Embedding*". Science. 290 (5500): 2323–2326

[25] Yixin Wang, Jan GM Klijn, Yi Zhang, Anieta M Sieuwerts, Maxime P Look, Fei Yang, Dmitri Talantov, Mieke Timmermans, Marion E Meijervan Gelder, Jack Yu, et al. Gene-expression profiles to predict distant metastasis of lymph-node-negative primary breast cancer. The Lancet, 365(9460):671–679, 2005.

[26] Daxing Xu, Bo Chen, Li Yu, Wen-an Zhang. "*Secure dimensionality reduction fusion estimation against eavesdroppers in cyber–physical systems*". https://www.sciencedirect.com/science/article/abs/pii/S0019057819304896

Image sources

- https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051
- Wagh, Sameer and Gupta, Divya and Chandran, Nishanth. SecureNN: 3-Party Secure Computation for Neural Network Training. https://eprint.iacr.org/2018/442.pdf, Figure 2 2019.