Optimizing convolutional neural networks in multi-party computation

Campmans, H.H.M.

Award date:
2018

Optimizing Convolutional Neural Networks in Multi-Party Computation

H.H.M. Campmans
Supervisor: Dr.ir. L.A.M. (Berry) Schoenmakers

April 26, 2018

Contents

Chapter 1

Introduction

Secure multi-party computation(MPC) allows mutual distrusting to evaluate a function with secret inputs. The output needs to be correct and the protocol needs to be secure, which means that no information other than the output of the function should be leaked. Two types of multi-party computation can be distinguished: multi-party computation based on homomorphic encryption and multi-party computation based on secret sharing. This thesis will only consider multi-party computation based on secret sharing, it assumes that the adversaries will follow the protocol and the function that is evaluated is publicly known.

A convolutional neural network (CNN) is a class of neural networks that is very succesful in analyzing visual data. The general idea of neural networks is to mimic the behavior of biological neural networks of animal brains. The input data is represented with a matrix and the network consist of layers. Every layer applies a linear operation on the data which results in a matrix. Every element of this matrix is put into an activation function. This nonlinear activation function mimics the nonlinear behaviour of the neuron and filters unrelevant information. The resulting matrix is put into the next layer and so on, until the output is obtained. The output can often be interpretted as a vector with scores. The higher the score, the more likely the data belongs to the corresponding image class. The approach appears to be quite succesful as CNNs are able to perform better than humans at the task of classifying handwritten symbols [4]. As larger neural networks often perform better, the state of the art neural networks are computationally expensive.

Convolutional neural networks are very popular in image recognition, which can be seen in the Image Large Scale Visual Recognition Challenge (ILSVRC) [5, 9, 10, 11, 16]. When such a service would be provided, a owner might get access to large amounts of possibly sensitive data. To protect this data it can be implemented in MPC. The goal of this study is to create a MPC version of a neural network and to optimize the corresponding computations. The first goal is to take a floating-point number based neural network and implement the inference phase (opposite of training phase) of the neural network in MPC. The second goal is to improve the calculation time. To this extent we consider both multi-party computation techniques and neural network techniques. A lot of information is known about both topics. But a lot less is known about the overlap, which can be called oblivious neural networks [14] or secure machine learning [15]. This offers space for further research

1.1 Thesis Outline

In Chapter 2 the preliminaries of the convolutional neural networks are discussed. The common techniques are explained an an example network is given. This example network is used during the thesis to evaluate techniques and hypotheses.

In Chapter 3 the fundamentals of Shamir secret sharing are discussed. Followed by the most important basic protocols and simplified versions of the more advanced protocols. This chapter should give an idea of the cost of the operations and how to measure the communication and computation complexity of a MPC protocol.

In Chapter 4 and 5 we discuss two important decisions that had to be made during the implementation of the example network. The current neural networks often work with floating point numbers. Since floating point operations are very expensive in MPC, Chapter 4 discusses the use of fixed point representations instead. Chapter 5 discusses the use of Fourier transforms that are a common technique to reduce the number of multiplications in convolutions. In MPC this is not necessarily an improvement.

In Chapter 6 we propose some new protocols that perform better than the old protocols with the same purpose. Most of these suggestions only work for the 3-party computation, or other settings that guarantee security for at most 1 adversary. The 3-party computation is an interesting case of MPC as 3 parties is the smallest number of parties to guarantee information-theoretic security against adversaries. Furthermore the number of sent messages in MPC grows quadratically in the number of parties and the local computations also grows with the number of parties. This makes 3-party computation the most efficient setting of MPC to resist adversaries. For this reason SHAREMIND [2] is built upon 3-party computation.

Chapter 2

Neural Network Preliminaries

2.1 Neural Networks

A neural network is a computing system that is inspired by the neural network of animals. The neurons of animals tend to fire a signal when they are stimulated in the right way. The distinction between firing on high input values Y and not firing on low input values Y forms an important basic feature for the computational model as well. The network is designed to perform a certain task, for example image classificiation. It is trained with labeled training data, i.e. the classifications are known, after which the network should be able te classify other images which are not in the training data. The general name for the phase after the training phase is called the inference phase. In the inference phase the network is supposed to be able to do the task it is designed for, so this could be classifying images, recognizing words in a voice recording or one of the other numerous applications of neural networks. Neural network are built in layers. Every layer consists of a set of linear operators followed by a set of nonlinear operators. The output of the first layer is the input for the next layer and so on. Often the linear operator is a matrix or a convolution operation. The convolution operation is explained on page 8. But in general the linear operators are represented by

$$Y = \sum(weight \cdot input) + bias$$

The weights and the bias are the network parameters, which are trained during the training phase. After applying the linear operators, Y is fed into the non linear operators. The set of non linear operators always contains a so-called activation function which mimics the behavior of a neuron. If the input Y is low, the neuron filters the information by passing on a 0 or another small value. If the input is a high value of Y, the neuron is "activated", in which case the neuron passes the information on to the next operation. In the activated state multiple levels of output exist such that the outputs from the active neurons can be compared. Examples of popular activation functions are the sigmoid function $\frac{1}{1+e^{-x}}$ and the Rectified Linear Unit (ReLU) $max(0,x)$, but there are many more and one can even make a custom activation function.

Another non linear operator which is used often is the max pooling operation, which takes the maximum element from a subset of data elements. The other data values will not be used anymore. The main purpose of this operation is to reduce the amount of data in the network, while maintaining the most relevant data. Average pooling is also an option, which has the benefits of reducing the dimensions, while being linear.

2.2 Training Phase

The first step of creating a neural network is designing the network, but more about that in the next section. Once the design is made it states the sizes of the input and the parameters and what operations are done during the inference phase of the neural network. A popular approach to train a neural network is stochastic gradient descent. Gradient descent is first described by Cauchy in 1847[3]. In gradient descent the network parameters are initially be filled with random values. By analyzing the gradients of the network it can be seen how changing the parameters would influence the output. Because the training data is labeled there is a clear view of direction in which the output should change to improve the network. The parameters get an update in the right direction and then the process starts all over again until some stop criterion is reached. This method is effective but not efficient as it is computationally expensive to involve all the training data in every iteration.

In stochastic gradient descent only a subset of the training data is used in every iteration. These subsets of the training data are taken randomly and in the end all data is used. This subset is still evaluated in the network and gradients determine in which direction the parameters should be altered to improve the network.

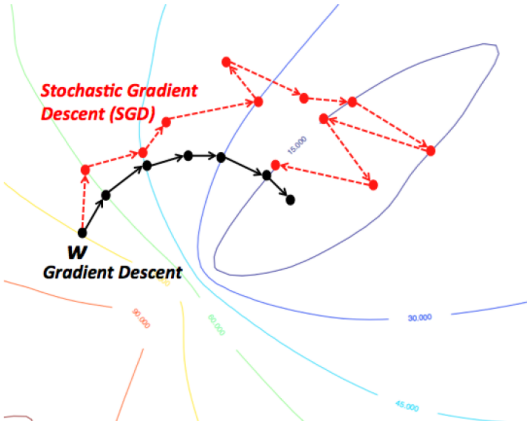


Figure 2.1: A visual comparison: Stochastic gradient descent is computationally less expensive.

To evaluate the performance of the trained network, we require access to more labeled data. The network can then be applied to the labeled data to get an estimation of how accurate the network performs in the inference phase.

2.2.1 Overfitting

Some problems might occur during the training of a network. One of them is overfitting. Overfitting means that the network performs better on the training data than on other data, which can be seen in the performance evaluation. A neural network should recognize properties of data classes in the right level of abstraction. The network should be optimized to recognize the underlying structure of the data set, not the specific data set itself.

2.2.2 Vanishing Gradient

During the training, weights are updated. The updates depend on gradients which are calculated backward from output up until the first layer of weights. Some activation functions have gradients in the range $[0,1]$. If the network has many layers and the resulting gradient is calculated through the chain rule, the product of these small numbers might result in very small derivatives. These vanishing gradients prevent the network from training properly. An activation function like the ReLU does not have this problem as it only has gradients 0 and 1.

2.3 Example Network

During the thesis one neural network is used to test hypotheses and the performance of protocols. This neural network is a convolutional neural network designed to classify handwritten decimal digits from the MNIST database. Every element from the database represents a element from $[0,1]^{28\times 28}$ with a corresponding label of the represented decimal digit. Every entry of the matrix represents a pixel from an image. If the matrix entry is 1, the corresponding pixel is black and if the matrix entry is 0 the corresponding pixel is white. Values between 0 and 1 represent shades of grey. The database consists of 65,000 labeled images of digits. 55,000 of these images are used during the training in tensorflow and the remaining 10,000 labeled images are used to evaluate the performance of the trained network. After explaining the necessary operations, the mathematical model of this neural network will be given in Algorithm 1 on page 11, which classifies 99.2% of the evaluation data correctly.

MaxPooling

The goal in a max pooling operation is to reduce the amount of data while maintaining the most import information. The size reduction has two benefits: A reduction in computational power to evaluate the network and reduction in the number of parameters needed for the network. As the previous section suggested, the most relevant signals are captured in the highest signal values. Therefore the max pooling operation passes on the highest value of a submatrix. For convenience we let the sizes of this submatrix be equal to the stride t, which implies that the submatrices do not overlap and the submatrices collectively overlap the original matrix.

Definition 2.3.1. Let n and m be the dimensions of the input matrix, k the number of input images that are processed simultaneously, r the number of intermediate result matrices we have per input image and t both the stride and the dimension of the submatrices.

$$\begin{aligned} &\text{MaxPooling}_t : \mathbb{R}^{k\times n\times m\times r} \to \mathbb{R}^{k\times \frac{n}{t}\times \frac{m}{t}\times r} \\ &\text{MaxPooling}_t(A) = B \\ &B_{k,i,j,l} = \max_{\delta_i,\delta_j\in\{0,\dots,t-1\}} \{A_{k,ti+\delta_i,tj+\delta_j,l}\} \end{aligned} \tag{2.1}$$

In our current model we use $t = 2$.

2.3.1 2D Convolutions

Neural networks require non linear parts, but a lot of work is done in the linear part as well. Matrix multiplication is a very commonly used linear operator to extract properties from data. In our case the problem with these matrices is that they do not capture the concept of locality: The spatial structure of the data is not exploited well enough. In an image two neighbouring pixels have a strong relation, and if they are much different it is likely that it displays the edge of an object or surface. This relation is not captured easily by a matrix. A convolution is exploiting the spatial structure of the data in images and sound recordings. For this reason they are commonly used, but not exclusively, in signal processing.

Convolutions are both defined for continuous functions as discrete mathematical objects. This section is about the discrete convolution for $2D$ objects. A $2D$ convolution is a linear operator which requires at least one input matrix and a kernel/filter. A convolution with one matrix and one kernel is visualized below:

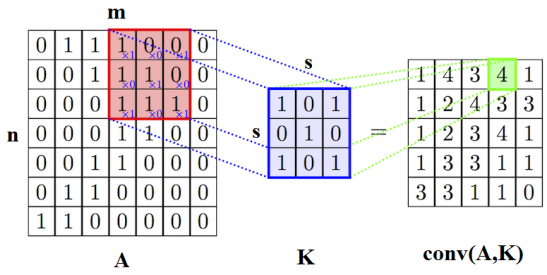


Figure 2.2: A is the input matrix and K is the Kernel. The red and the blue square get multiplied elementwise and the sum of all these products is written in the green square. If the red square shifts, the green targets square shifts accordingly.

The operation illustrated in the image above is repeated r times, and the resulting matrices are summed. For bookkeeping purposes A and K are higher dimensional, but the objects can be seen as double indexed array of matrices.

Definition 2.3.2. Let n and m be the dimensions of the input matrix, k the number of input images we are processing simultaneously, r the number of intermediate result matrices we have per input image, s the size of the square 2D kernel and v the number of intermediate result matrices we have after applying the operation. Further we have $s \equiv 1 \mod 2$. We define the operation as follows

$$\begin{aligned} &\text{conv} : \mathbb{R}^{k\times n\times m\times r} \times \mathbb{R}^{s\times s\times r\times v} \to \mathbb{R}^{k\times n\times m\times v} \\ &\text{conv}(A,K) = B \\ &B_{b,i,j,t} = \sum_{\delta_i=-\frac{s-1}{2}}^{\frac{s-1}{2}} \sum_{\delta_j=-\frac{s-1}{2}}^{\frac{s-1}{2}} \sum_{q=0}^{r-1} A_{b,i+\delta_i,j+\delta_j,q} \cdot K_{\delta_i,\delta_j,q,t} \end{aligned} \tag{2.2}$$

In the case that $i+\delta_i$ or $j+\delta_j$ is out of bound we consider $A_{b,i+\delta_i,j+\delta_j,q} = 0$

The indices i and j are referring to the position of the filter relative to the matrix as depicted with the red square in Figure 2.2. The index b is the index of the input image we are working on. The general definition in (2.2) can be seen as applying Figure 2.2 on $k \cdot r \cdot v$ different pairs (A,K).

2.3.2 The Multilayer Convolutional Network

The overall model is as follows

$$\text{MMCN} : [0,1]^{k\times 28\times 28\times 1} \to \{0,1,...,9\}^k$$

The parameters of layer d are denoted with $W^{(d)}$ and $b^{(d)}$. The W's contain the convolution kernels/filters and matrices and the b's contain the bias vectors. For reference we define $A^{(d)}$ to be the intermediate result after layer i. In reality the intermediate results do not need to be saved.

Algorithm 1 The MNIST Multilayer Convolutional Network

layer1: $A \in \mathbb{R}^{k\times 28\times 28\times 1}, W^{(1)} \in \mathbb{R}^{1\times 5\times 5\times 32}$
  $A \leftarrow conv(A, W^{(1)})$
  $A_{l,i,j,m} \leftarrow A_{l,i,j,m} + b^{(1)}_m$
  $A \leftarrow \text{MaxPool}_{2,2,2}(A)$
  $A^{(1)} \leftarrow \text{ReLU}(A)$
layer2: $A^{(1)} \in \mathbb{R}^{k\times 14\times 14\times 32}, W^{(2)} \in \mathbb{R}^{32\times 5\times 5\times 64}$
  $A \leftarrow conv(A^{(1)}, W^{(2)})$
  $A_{l,i,j,m} \leftarrow A_{l,i,j,m} + b^{(2)}_m$
  $A \leftarrow \text{MaxPool}_{2,2,2}(A)$
  $A^{(2)} \leftarrow \text{ReLU}(A)$
layer3: $A^{(2)} \in \mathbb{R}^{k\times 7\times 7\times 64}, W^{(3)} \in \mathbb{R}^{1000\times 3136}$
  reshape $A^{(2)}$ to $\mathbb{R}^{(7\cdot 7\cdot 64)\times k}$
  $A \leftarrow W^{(3)} \cdot A^{(2)}$
  $A_{i,j} \leftarrow A_{i,j} + b^{(3)}_i$
  $A^{(3)} \leftarrow \text{ReLU}(A)$
layer4: $A^{(3)} \in \mathbb{R}^{1000\times k}, W^{(4)} \in \mathbb{R}^{10\times 1000}$
  $A \leftarrow W^{(4)} \cdot A^{(3)}$
  $A^{(4)}_{i,j} \leftarrow A_{i,j} + b^{(4)}_i$
  $A \in \mathbb{R}^{10\times k}$
return $\text{maxArg}(A)$

$A^{(4)}$ can be interpretted as a matrix that states scores $A^{(4)}_{i,j}$ which represent the score for input image j and digit i. So the higher the score, the more likely the image with index j represents the decimal digit i. The last line of the algorithm picks the highest of these 10 scores.

Chapter 3

Cryptographic Preliminaries

In secure multi-party computation multiple parties cooperate to evaluate functions. The goal is to achieve correct results while nobody learns anything beyond the output of the computation. An honest party is a party which follows the protocol and does not try to gain more information about the calculation than the output of the calculation. Adversaries try to learn information about the computation except the output. In the passive security case we only consider semi-honest adversaries, which are adversaries that follow the protocol, but use any information in their view to learn secret information. In the active security case adversaries might deviate from the protocol to learn information or impede the calculation. This thesis only considers the passive security case.

There are multiple basic principles which can be used to realize the properties above. For example additive secret sharing, Yao's garbled circuits [19], Shamir secret sharing [17] and homomorphic encryption. In this thesis Shamir secret sharing is used. For secret sharing in general, parties share their data among the participating parties. Every qualified subset of these parties is able to reconstruct all the secrets. Any subset of parties which is not qualified should be unable to learn anything about the secrets. If the total number of parties is n, any subset of size at most t parties is unqualified and any subset of size $t+1$ or larger is qualified, we have a (t,n)-secret sharing scheme. Shamir secret sharing is such a (t,n)-secret sharing scheme. The property that t parties learn nothing and $t+1$ players know everything is referred to as the threshold property.

Shamir secret sharing is able to keep all the secrets secret if there are at most t semi-honest adversaries involved. To be able to multiply in Shamir secret sharing the inequality $n \geq 2t+1$ needs to hold. And since a smaller t would imply that more people need to be honest, it is common to have t as large as possible to get the best security against adversaries. Hence $n = 2t+1$ and $n = 2t+2$ are very common, which means that Shamir secret sharing is secure as long as at least $\lceil \frac{n+1}{2} \rceil$ parties are honest, which we call an honest majority. From this point onward we assume to have such an honest majority.

To make a clear distinction between the public values and the secret values, there are brackets around secret shared values, like $[x]$ opposed to the publicly known x. Operations which only require public data are said to be "in the clear" to emphasize that these operations are done with public operations, which are much cheaper because no communication between parties is required. In the rest of this chapter we explain the mathematical concept behind Shamir secret sharing after which we explain the most important protocols from the Shamir secret sharing arithmetic.

3.1 Shamir Secret Sharing

In Shamir secret sharing secrets are represented by polynomials of degree t in $\mathbb{F}_p[x]$. A secret $[a]$ is stored in a polynomial $p_a(x)$ with $p_a(0) = a$. The polynomial is of the form

$$p_a(x) \equiv a + \sum_{j=1}^{t} \alpha_j x^j \bmod p$$

in which the coefficients α_j are generated as random numbers in $\mathbb{Z}_p$. Note that the degree of $p_a(x)$ is at most t. If a is shared, a polynomial $p_a(x)$ is generated and each party P_i, with $i \in \{1,2,...,n\}$, receives $p_a(i)$ as their share of the secret. Note that $p_a(x)$, or more specific $p_a(0)$, can be reconstructed by Lagrange interpolation if $t+1$ or more parties cooperate. With less than $t+1$ cooperating parties combining their knowledge, all values of $a \in \mathbb{F}_p$ are equally likely to be represented based on the knowledge of these parties. Hence it is information-theoretically impossible to learn anything about the secret with less than $t+1$ shares.

VIFF is a framework that uses Shamir secret sharing protocols to implement MPC. VIFF, the Virtual Ideal Funcionality Framework, is a python based framework in which secure multi-party computations can be specified. TUeVIFF is an improved version of VIFF with enhanced runtime performance. In this thesis all implementations of MPC are built with TUeVIFF. From this point onward we will refer to TUeVIFF with VIFF.

3.1.1 Creating and Opening Shares

The protocols in this subsection can all be found in the thesis [6]. As shares are the building blocks of Shamir secret sharing, this section shows how to create and open them. First we show how to create a shared secret out of private data. The share of party P_i that corresponds to the secret $[s]$ is denoted with $[s]_i$.

Protocol 2 [s] $\leftarrow$ SShare(s,t,n)

1: **pick** $\alpha_1,...,\alpha_t \in_R \mathbb{F}_p$
2: **for each** $j = 1,...,2t+1$ **do**
3: $\quad [s]_j \leftarrow s + \sum_{\ell=1}^{t} \alpha_\ell j^\ell$
4: $\quad$ **send** $[s]_j$ **to party** P_j
5: **return** $[s]$

When a secret s is opened, every party P_i has a share $p_s(i)$ and the value $[s] = p_s(0)$ needs to be reconstructed. Lagrange interpolation states that

$$p_s(0) = \sum_{i \in D} p_s(i) \prod_{j \in D, j \neq i} \frac{-j}{i-j} \tag{3.1}$$

can be used to reconstruct this value with $D \subseteq \{1,2,...,2t+1\}$ such that $|D| > \deg(p_s)$ holds. Opening a secret can be done by any such qualified subset D. Note that the products $\prod_{j \in D, j \neq i} \frac{-j}{i-j}$ can be precomputed.

Protocol 3 x $\leftarrow$ SOpen$(D,[x])$

1: **for each party** P_i **with** $i \in D$ **send** $[s]_i$ **to all parties**
2: $s = \sum_{i \in D} [s]_i \prod_{j \in D, j \neq i} \frac{-j}{i-j}$
3: **return** s

3.2 Native Shamir Secret Sharing Arithmetic

The protocols in this section can all be found in the thesis [6] With Shamir secret sharing it is possible to perform addition without communication. When $[x]+[y]$ is calculated, every party P_i adds $p_x(i)+p_y(i)$ to find $p_{x+y}(i)$ locally. Also multiplication of a public number with a secret does not require any communication. If a public value c is to be multiplied with the secret $[x]$, then every party P_i calculates $c \cdot p_x(i)$ to find $p_{c \cdot x}(i)$. On the other hand performing inner products does require communication. Any multiplication can be written as an inner product and is equally expensive in terms of communication. If the inner product of the vectors with secrets $[\bar{x}]$ and $[\bar{y}]$ are calculated, then every party computes $p_{\bar{x} \cdot \bar{y}}(i) = \overline{p_x(i)} \cdot \overline{p_y(i)}$ locally. The problem with this result is that it is a polynomial of degree $2t$, which cannot be opened by every qualified subset. Hence Protocol 4 shows how the degree of the polynomial is decreased using the properties of (3.1).

Protocol 4 [c] $\leftarrow$ Inner$([\bar{x}],[\bar{y}])$

1: **for each party** $i = 1,...,2t+1$ **do**
2: $m_i \leftarrow [\bar{x}]_i [\bar{y}]_i$
3: $[m_i] \leftarrow$ SShare(m_i, t, n)
4: $[c]_i \leftarrow \sum_{j=1}^{2t+1} ([m_j]_i \prod_{\ell=1, \ell \neq j}^{2t+1} \frac{-\ell}{j-\ell})$
5: **return** [c]

We use the term *invocation* as a unit, which is defined by the amount of communication that is used in Protocol 4. Note that a reshare costs exactly one invocation. An invocation implies that every party is sending one value of $\mathbb{Z}_p$ to every other party. In MPC 3 factors can play a main role for fast function evaluation: computations, bandwidth and delay. For the computational complexity we use the same techniques as in the clear. For bandwidth we measure the number of messages that is sent over the channels between parties. The invocation is a unit for bandwidth. It takes a while for all messages to arrive. Therefore it is not efficient to wait for every message to arrive and instead we send as much messages as possible in parallel. However, some operations can only be started after the results of previous operations have been received. These operations are put into a buffer such that they will be executed as soon as all the prerequisite shares are in place. Meanwhile the processor can continue performing local operations for different operations. To quantify the delay that happens because of these depencies, the round complexity is introduced. 1 round is a set of invocations which can be sent simultaneously because the content of the invocations do not depend on each other. The results of the first round can be used in the second round (or later) and so on. For example, calculating a 4th power requires two consecutive multiplications(squaring twice), which costs 2 rounds and 2 invocations, while calculating $[x] \cdot [y]$ and $[y] \cdot [z]$ simultaneously requires only 1 round and 2 invocations.

3.3 Protocols Based on Bit Representation

The two protocols above are closely related to the behaviour of $\mathbb{F}_p[x]$. The comparison, LSB and truncations require more advanced techniques, which we will explain in this subsection. Truncations are useful for the fixed-point number representation and comparisons are an essential building block for the non-linearity of the neural networks.

This section is working towards presenting a protocol to find the least significant bits, which requires pseudo random bit generation. For the generation of pseudo random numbers in VIFF there is a convenient trick in place to save a lot of communication. The trick is called pseudo random secret sharing (PRSS) [6]. PRSS requires a setup phase in which every party locally computes a random number which is shared to the other parties. This random number will be used as a seed for the generation of more pseudo random numbers by using local computation only. It should be mentioned that this local computation is doing a for-loop over all the t sized subsets of the n participating parties that contain the computing party. Since $\binom{n}{t}$ is growing superexponentially for $t \approx \frac{n}{2}$, this is only efficient for small n. This section does not go into the detail of PRSS, but for the reader it is sufficient to know that players can generate uniformly random field elements of $\mathbb{F}_p$ without communication. A shared k-bit integer can also be generated without communication, but the distribution is non-uniform: It is the sum of $\binom{n}{t}$ uniform distributions. These uniform distributions have upper bounds which are chosen such that the sum of these random values cannot exceed k bits.

The following method of generating a single random bit is based on calculating a (nonzero) square of a pseudo random field element that is subsequently opened. When calculating the principal square root $\sqrt{r^2} = (r^2)^{(q+1)/4} \bmod q$ in the clear we have a probability of $\frac{1}{2}$ to find r and a probability of $\frac{1}{2}$ to find $-r$. This property is exploited to construct a secret $[b] \in \{0,1\}$ with corresponding probabilities. Note that raising a number to the power $\frac{q+1}{4}$ in the clear has a relatively high computational complexity.

Protocol 5 [b] $\leftarrow$ PRandBit

1: **do**
2: $\quad [r] \leftarrow \text{PRandFld}(\mathbb{Z}_p)$
3: $\quad u \leftarrow \text{MulPub}([r],[r])$
4: **while** u=0
5: $v \leftarrow u^{-(q+1)/4} \bmod q$
6: $[b] \leftarrow (v[r]+1)2^{-1} \bmod q$
7: **return** $[b]$

The random bits created with Protocol 5 can be used to find the least significant bit(LSB) of a secret. The random bits can be used to map random self-reducible problems to other instances of the same problem. An example of a random self-reducible problem is solving a discrete logarithm. If we look for x such that $g^x \equiv y \bmod p$, which is a hard problem to solve, then we can map the problem to the problem $g^{\tilde{x}} \equiv \tilde{y} \bmod p$ where $\tilde{y} = y \cdot g^s$ and $\tilde{x} = x + s$. The new problem can be sent to another party. This other party can compute $\tilde{x}$, but cannot deduce x as it does not know s. If the answer is sent back to us we can calculate $x = \tilde{x} - s$. In this fashion we can map random self-reducible problems to other instances of the same problem. In MPC this technique is interesting as no information should be leaked about the solution of the original problem, but the transformed problem can be solved in the clear. The parameter s that is used to transform the problem is a shared value $[s]$.

Applying this technique on the LSB gives the following approach. The original problem is finding $(x \bmod 2)$, which will be altered into finding $(x + 2 \cdot R + b_0 \bmod 2) = \text{LSB}(x) \oplus b_0 = c$ which can be solved in the clear. The actual answer is $\text{LSB}(x) = c_0 \oplus [b] = c_0 + [b] - 2c_0[b]$. This notation for the XOR operations holds if $c_0, b \in \{0,1\}$.

Protocol 6 [b] $\leftarrow$ LSB($[x]$)

1: $[r_0] \leftarrow \text{PRandBit}(\mathbb{Z}_p)$
2: $[r'] \leftarrow \text{PRandInt}(\mathbb{Z}_p, k + \kappa - 1)$
3: $c \leftarrow \text{SOpen}([x] + [r_0] + 2[r'])$
4: $c_0 \leftarrow (c \bmod 2)$
5: $[b] \leftarrow c_0 + [r_0] - 2c_0[r_0]$ $\quad [b] = c_0 \oplus [r_0]$
6: **Return** $[b]$

The secret input of Protocol 6 represents an integer which has an absolute value smaller than 2^{k-1}. This implies that $2^{k-1} + x$ is always a non-negative number in the range $[0, 2^k]$. This transformation is important to define what we mean with the parity of negative numbers. $p - 1$ is even and -1 is odd, but $p - 1 \equiv -1 \bmod p$. In practice we add $2r'$ which is bigger than 2^{k-1} with overwhelming probability. This means that we can ommit the $+2^{k-1}$ and still know for sure that we are working with a positive number all the time, such that -1 is an odd number. To prevent the leaking of information it is relevant that $p > 2^{k+\kappa+\log(n)}$ where κ is a security parameter. Note that if $x + r_0 + 2r' > p$ then c would be the $\bmod p$ reduced version of $x + r_0 + 2r'$, which would ruin the property that $x + r_0 \equiv c \bmod 2$. Therefore r' is smaller than $2^{k+\kappa-1}$ while $p > 2^{k+\kappa+\log(n)}$ such that $c := 2^{k-1} + x + r_0 + 2r' < p$. Opening c leaks statistical information about x, but the statistical information gained by an adversary is negligible since r' is picked from a larger range than the range x lives in. The security parameter κ determines the statistical security in this process. This κ quantifies the amount of overhead in the system, which is the number of bits we can add to any meaningful secret x before we obtain overflow $\bmod p$.

We now present a protocol for truncation. Truncation is the operation in which the f least signficant digits or bits are removed from the secret, which behaves like $\lfloor \frac{(x \bmod p)}{2^f} \rfloor$ in which we talk about the division operation from $\mathbb{R}$. This should not be confused with the division operation from $\mathbb{F}_p$. However, if a secret x is divisible by 2^f the division of $\mathbb{F}_p$ behaves the same way as the truncation. The protocol below gives the exact right answer and is not optimal in terms of round complexity. It serves as an example to illustrate the idea of how truncations can be calculated with Shamir secret sharing.

Protocol 7 [y] $\leftarrow$ simpleTrunc$([x], f)$

1: $[y] \leftarrow 2^k + [x]$
2: **for** $i = 1$ **to** f **do**
3: $\quad [y] \leftarrow ([y] - \text{LSB}([y]))2^{-1}$
4: **return** $[y] - 2^{k-f}$

The optimal implementation which is used in VIFF is using 4 rounds and $4f + 2$ invocations [6]. If a user is willing to accept a rounding error such that the output might be $trunc(x, f) + 1$, it can be done with with 2 rounds and $f + 1$ invocations [6]. The probability that this rounding error occurs is $\frac{(x \bmod 2^f)}{2^f}$, which is $\pm\frac{1}{2}$ if the f least significant bits are uniformly distributed.

The following protocol demonstrates how a LSB protocol can be used to calculate a less than zero (LTZ) comparison:

Protocol 8 [b] $\leftarrow$ simpleLTZ$([x])$ returning $[x \leq 0]$

1: $[y] \leftarrow 2^k + x$
2: $[z] = \text{LSB}([y])$
3: $[y] = ([y] - [z])2^{-1}$
4: **for** $i = 1$ **to** $k - 1$ **do**
5: $\quad [b] \leftarrow \text{LSB}([y])$
6: $\quad [z] \leftarrow [z] + 2^i[b]$
7: $\quad y \leftarrow ([y] - [b])2^{-1}$
8: **return** $([z] - [x])2^{-k}$

More complicated techniques can be used to lower the cost of the LTZ [6] comparison to $\log(k) + 2$ rounds and $3k - \log(k) - 1$ invocations with statistical security [6], where k is an integer such that $x \in [-2^k, 2^k)$. Note that Protocol 8 also has statistical security as the LSB has no perfect security.

Chapter 4

Fixed-Point Precision

When a neural network performs well in the clear, can it simply be converted into an algorithm suitable for MPC? How do computations with non-integer values need to be handled in MPC? The goal of this chapter is to show the reader the best way to represent real numbers in MPC and why this is the best way. In the clear, floating-point numbers (floats) are commonly used in basic implementations of neural networks. Many computers are equipped with hardware that is optimized for floating-point arithmetic. In most MPC frameworks, floats are not supported and if supported, both addition and multiplication come with high costs like in [12].

The use of floats has the benefit that the maximal rounding error scales with the magnitude of the number. The most common alternative to floats is the fixed-point arithmetic, which has rounding errors of fixed size. Working with fixed-point numbers requires more precaution to make sure that rounding errors do not dominate the calculation. However, using a float representation in Shamir's secret sharing scheme is computationally expensive. This chapter compares the computational complexity of these two representations after which some further optimizations are investigated.

4.1 Floating-Point Numbers

Non-negative floats are represented as $s \cdot r^e$, which have a significand s, a radix r and an exponent e. The radix is fixed and need not be stored. In the case of binary representations this radix is 2, so $r = 2$ from this point onward. The bit-size of s and e are fixed. These bit-sizes are parameters that might differ between computers or software packages, but they are fixed within one run of a calculation. To achieve the best possible precision, the significand is normalized, in the sense that the most significant bit of s is a 1 after which the radix point will be put such that $s \in [1,2)$.

For this reason the most significant bit is often omitted in bit representation, such that only the fractional bits of the significand is stored. Hence a significand with a bit representation of 011001 represents a significand of $1 + \frac{0}{2} + \frac{1}{4} + \frac{1}{8} + \frac{0}{16} + \frac{0}{32} + \frac{1}{64}$. An example of a floating-point format is the half-precision floating-point, which occupies 16 bits per number, consisting of 1 sign bit, 5 bits for the exponent and 10 bits for the fractional part of the significand. The half-precision floating-point format is designed to store floats with relative low precision, which functions as a lower bound for the precision in this section.

4.1.1 Addition

Assume two positive numbers $s_1 \cdot 2^{e_1}$ and $s_2 \cdot 2^{e_2}$ are added. This results in

$$s_1 2^{e_1} + s_2 2^{e_2} = \text{normalize}(s_1 + s_2 \cdot 2^{e_2 - e_1}) \cdot 2^{\max(e_1, e_2) + c} \tag{4.1}$$

which is written in float representation with the carry $c \in \{0,1\}$ and $c = 1 \Leftrightarrow s_1 + s_2 \cdot 2^{e_2 - e_1} \geq 2^{1+\max(0, e_2 - e_1)}$.

Evaluating the right hand side of (4.1) in MPC gives rise to some problems:

- $2^{e_2 - e_1}$: Raising a public constant to the power of a secret number is a computationally expensive operation.
- normalize: Let p_s be the bit-size of s. The normalize-function is resizing the input, such that the p_s most significant bits are the bit representation of s. Any other bits are removed with the truncation operation. Also the number of bits that have to be truncated is secret in this case, which makes it even more expensive than truncation as described in Section 3.3.
- $\max(e_1, e_2)$: This operation requires at least a comparison, which is computationally expensive.

This implies that addition of two secret shared floats is at least as hard as the comparison which is involved in $\max(e_1, e_2)$. If the implementation of Section 3.3 would be used for this comparison, this would cost $k + 3$ invocations, where k is the number of bits used to express exponents. This cost is high compared to the free addition of finite field elements in Shamir secret sharing.

4.1.2 Multiplication

With respect to multiplication, we can write the product of two floats as

$$s_1 2^{e_1} \cdot s_2 2^{e_2} = \text{normalize}(s_1 \cdot s_2) 2^{e_1 + e_2 + c}$$

The main problem of floating-point multiplication lies in the normalization. To perform the truncation in the normalization, it needs to be known how many least significanbt bits should be removed. This number of bits depends on the question whether $s_1 \cdot s_2 \geq 2$ holds.

- comparison in normalize: For the float multiplications the mentioned comparison $s_1 \cdot s_2 \geq 2$ would cost $k + 3$ invocation. If the comparison is performed after the truncation, k is the number of bits used to express the significand, otherwise k would even be larger. With half-precision floating-point this results in $k = 10$ and hence 13 invocations.

There are more downsides which could be addressed such as handling different representations of zero. But this section should give sufficient insight to see that any implementation of floats in Shamir's secret sharing scheme has an inherent expensive addition and multiplication operation.

Conclusion

Shamir secret sharing has the finite-field addition and inner product as cheap native operations which work in one-to-one correspondence with integers if numbers remain smaller than the modulus. Other operations are supported, but are often computationally more expensive. The floating-point arithmetic makes addition and multiplication expensive as it relies on functions which are not native in Shamir secret sharing. The computation requires secure access to the bits of the binary representation, which is expensive in Shamir secret sharing. As additions and multiplications occur many times in a convolutional neural network, the high costs are prohibitive for floating-point arithmetic.

4.2 Fixed-Point Numbers

As alternative to floating-point, fixed-point arithmetic can be considered. A fixed-point number has the format $x = z \cdot 2^{-f}$ in which f is called the resolution. The integer z is a number of at most k bits from which the least significant f bits represent the fractional part of the fixed-point number. The number $e = k - f$ is called the range and every fixed-point number has a value $-2^{e-1} < x \leq 2^{e-1}$. When the integers z are used as secrets in Shamir secret sharing, addition remains a local operation:

$$z_1 \cdot 2^{-f} + z_2 \cdot 2^{-f} = (z_1 + z_2) \cdot 2^{-f} \tag{4.2}$$

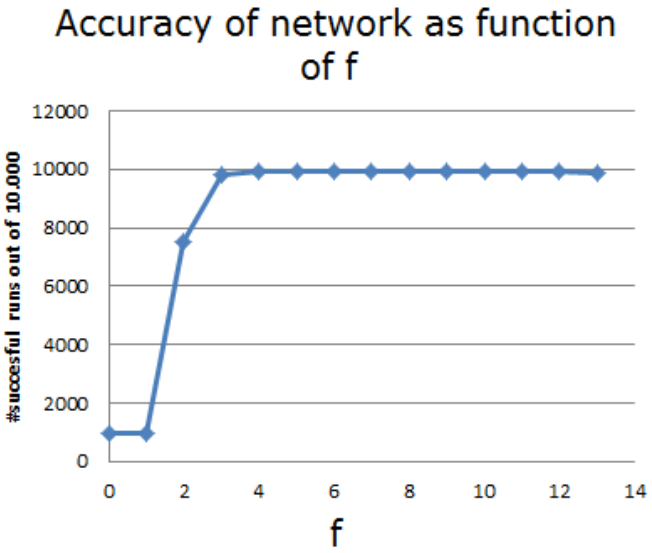
And for multiplication the following formula holds

$$(z_1 \cdot 2^{-f}) \cdot (z_2 \cdot 2^{-f}) = \text{Trunc}_f(z_1 \cdot z_2) 2^{-f} \tag{4.3}$$

The only non-trivial part of this multiplication is one truncation and the number of bits that need to be removed is known. In both the addition and the multiplication operations there is no mechanism to prevent that $|z_1 + z_2| > 2^{e-1}$, so the programmer needs to know bounds of the numbers he is working with, such that this z stays within the allowed range.

4.2.1 Neural Network Accuracy with Fixed-Points

In the initial model floats are used in the clear and as described in the previous section, these values are rounded to fixed-point numbers. From this conversion can be concluded that the larger f, the more significant bits there are to work with. Note that the neural network contains comparisons within the ReLU and Maxpooling operation. The cost function of the comparison depends on the number of bits of the compared numbers. So the higher f, the more expensive the comparisons will be and the overall computation cost will therefore increase. On the one hand a large f implies an expensive computation and on the other hand a small f might impair the ability of the neural network to properly classify the data. To understand the effect of f on the accuracy of the neural network, the network is implemented in the clear with resolution f. In the experiment a run of the protocol is successful if the neural network classifies a digit correctly. In this test the numbers are truncated after every multiplication. If the truncations would be postponed or skipped, the truncations would lead to a smaller rounding errors. For different values of f the success rates are shown below.



(a) figure

f	2^f	succes
0	1	980
1	2	980
2	4	7536
3	8	9809
4	16	9910
5	32	9907
6	64	9915
7	128	9912
8	256	9912
9	512	9912
10	1024	9913
11	2048	9913
12	4096	9913
13	8192	9913

(b) table

Figure 4.1: The accuracy of Algorithm 1 as function of the resolution: from $f = 4$ the accuracy stays fairly constant

Tensorflow scores 9913 successful runs out of the $10,000$ test data samples with the same network parameters. Note that the values $f \leq 1$ have a succes rate of approximately 10%, which suggests it is equally smart as random guessing the decimal digit class. Furthermore we conclude from Figure 4.1 that not much precision is needed for the neural network to perform well.

These results are not new in the world of neural network. Dedicated hardware is already being built under the assumption that 8 bits are enough for the inference phase of neural networks. So called Tensor Processing Units (TPUs) [7] are developed by Google and use only 8 significant bits. These TPUs are only meant for the inference phase of a neural network and the paper states that virtually all training is still done with floating points nowadays.

4.2.2 Truncation Management

The benefit of fixed-point numbers over floating-point numbers is that no comparison needs to be done in the multiplication. The truncation that needs to be done in the fixed-point number case is cheaper than a comparison, but still more expensive than the native Shamir secret sharing multiplication. For this reason it is investigated wether it is useful to postpone truncations. By postponing a truncation (at every multiplication of that layer) the resulting output would be a factor 2^f off. As long as no overflow occurs and the programmer anticipates on this extra factor 2^f correctly, the program functions properly and there are even fewer intermediate rounding errors. After the linear operations, before the comparisons of the activation functions truncation is an option in every layer of the neural network. The following strategies are considered:

- The numbers are truncated in every layer, doing this gives a fixed point implementation of the neural network.
- The truncations are postponed. This means we do truncate in some layers, but not in all of them. This way we save costs of the truncation operations, while in the long run most of the abundant bits are removed.
- The numbers are never truncating at all. In this approach the bit length of the secrets is never reduced. And as larger secrets require a larger modulus, this option requires the largest modulus p.

By experiment we found out that it is the fastest for our example network to use no truncations at all. Evaluating 2 images in a 3 party setting took 25 minutes and 15 seconds with truncations at every layer. Repeating the same experiments with no truncations at every layer resulted in 22 minutes and 38 seconds. But since this network has not many layers we try to extrapolate the numbers of our network to estimate the results for deeper neural networks.
$f = 4$

Theoretical experiment
The truncations only influence the speed of the comparisons as the performance of all other protocols that are used in Algorithm 1 are not depending on the bit size. If all the fixed-point numbers get f bits removed in a layer, the output is roughly divided by 2^f, which factor propagates through the rest of the neural network. Hence all the following comparisons can become cheaper if the upper bound for the secrets is decreased by f bits.

A truncation costs 2 rounds, $f+1$ invocations and f exponentiations, with f the resolution of the fixed-point numbers. A comparison costs $1+\lceil \log_2(\ell+2) \rceil$ rounds, $2\ell+2$ invocations and ℓ exponentiations, with ℓ an upper bound on the number of bits of the secret.

Facts from the Example Network

To get a view on how much costs we can save, we counted the number of truncations and comparisons in our network:

layer	command	operation	number of calls	
1	conv	truncation	$25,088$	$(28 \cdot 28 \cdot 32)$
1	maxPool+ReLU	comparison	$25,088$	$(14 \cdot 14 \cdot 4 \cdot 32)$
2	conv	truncation	$12,544$	$(14 \cdot 14 \cdot 64)$
2	maxpool+ReLU	comparison	$12,544$	$(7 \cdot 7 \cdot 4 \cdot 64)$
3	matrix product	truncation	$1,000$	(1000)
3	ReLU	comparison	$1,000$	(1000)
4	matrix product	truncation	10	(10)
4	ArgMax	comparison	9	$(10-1)$

As the cost of the comparison operation depends on the bit size of the secrets, we also calculated an upper bound for the bit sizes of the intermediate results after every layer. In the following upper bound it is assumed that truncations are used in every layer.

after layer	maximum number of bits
1	$2+f$
2	$8+f$
3	$16+f$
4	$22+f$

From these observations we conclude some general rules:

- If a layer uses truncations, it uses the same number of truncations as the number of comparisons that are used in the activation function(s) of the same layer.
- The number of bits of the secrets grows throughout the layers. Note that some data is easy to classify and some data is not easy to classify. The data that is easy to classify probably results in larger numbers in the intermediate results. If more than f bits would be removed, we might remove the important bits from the data that is hard to classify. This is a downside of the fixed-point representation.

This information is already enough to conclude what strategy is optimal in terms of invocations and local computation (measured by the number of exponentiations).

- For the number of invocations it is the best to truncate at every layer.
  If we perform a truncation in a layer, it costs us $f+1$ invocation per number of the intermediate result. Because we did the truncation, the comparison that follows within the same layer costs $2(\ell-f)+2$ invocations per number instead of $2\ell+2$. Therefore it is more efficient to truncate everywhere in terms of invocations.
- For the number of exponentiations it is the best to truncate at every layer.
  If we perform a truncation after a layer, it costs us f exponentiations per number. The comparisons within the same layer costs $\ell-f$ exponentiations instead of ℓ. But since the later layers also benefit from the truncation, overall it is beneficial to do the truncation.

- For the number of rounds it is more complicated:

For this purpose we extrapolated our experimental network to a network with 9 fictional layers. The fictional upper bound for layer i is $10+2i$ bits plus the $f = 8$ bits per layer. If truncations are done every layer this results in $10+2i+f$ significant bits after layer i. If no truncations are done every layer, this results in $10+(2+f)\cdot i$ significant bits at layer i. In the experiment 4 scenarios are evaluated: truncating every where, truncating every 2 layers, truncating only halfway the neural network and never truncating.

The number of round induced by the comparisons are shown below

trunc in:	every layer		every 2 layers		only halfway		never	
layers	trunc(f)	comp	trunc(2f)	comp	trunc(5f)	comp	trunc	comp
1	2	5	0	6	0	6	0	6
2	2	5	2	5	0	6	0	6
3	2	5	0	6	0	7	0	7
4	2	6	2	6	0	7	0	7
5	2	6	0	6	2	6	0	7
6	2	6	2	6	0	6	0	8
7	2	6	0	6	0	7	0	8
8	2	6	2	6	0	7	0	8
9	2	6	0	7	0	7	0	8
total	69		62		61		65	

Table 4.1: The number of rounds used in every layer using multiple truncation strategies.

This implies that postponing truncations is useful for the round complexity of a neural network.

This implies that fixed-points should be used with truncations. In terms of round complexity it is better to postpone some truncations and truncate once every few layers, while the local complexity and number of invocations are optimal by truncating in every layer.

Chapter 5

The MPC Complexity of Fourier Transforms

The convolution, as defined in (2.2), has been an important building block for image recognition since LeNet5 [13] in 1994 and even before that it was used in many other applications. Hence a lot of people invested time in optimizing the computational complexity of the convolution operation, which makes it likely that it cannot be improved much more in the future. The most popular approach is to use Fourier Transforms to reduce the costs of the convolution. The main goal is to reduce the number of multiplications, which is the bottleneck for convolutions in the clear. In this chapter we evaluate whether it is profitable to use this technique in MPC as well.

The Fourier transform and the inverse are both linear transforms that map data from the spatial to the frequency domain or back. As we are interested in the transformation of matrices, we use the Discrete Fourier Transform(DFT). This transforms can be used to reduce the number of multiplications in the convolution using the following property:

$$\begin{aligned} &A, K \in \mathbb{R}^{m \times m} \vee A, K \in \mathbb{R}^{m} \\ &\mathrm{DFT}(A) \circ \mathrm{DFT}(K) = \mathrm{DFT}(\mathrm{conv}(\mathrm{A}, \mathrm{K})) \end{aligned} \tag{5.1}$$

The convolution has been reduced to the pointwise/Hadamard product denoted with $\circ$. In CNNs the filter K is typically smaller than the matrix A. Therefore the filter $K \in \mathbb{Z}_p^{s \times s}$ can be extended to $\mathbb{Z}_p^{m \times m}$ by adding columns to the right and rows to the bottom filled with 0s before transforming it into the frequency domain. This would also work for non square matrices, requiring two roots of unity, but for the sake of simplicity we will only consider square matrices.

5.1 Definition of DFT in $\mathbb{Z}_p$

The Fourier transform always uses roots of unity. It is common to use a complex root of unity for a Fourier transform, but in our case we use the Number Theoretic Transform(NTT) [1]. The NTT uses a root of unity $\alpha \bmod p$ with the property $\alpha^m \equiv 1 \bmod p$ and $\alpha^j \neq 1 \bmod p \; \forall j \in \{1, 2, \ldots, m-1\}$. As $m | p - 1 = \phi(p)$ must hold for such an α to exist, this gives an extra constraint on the used prime modulus p.
In the following definition the x denotes a vector in the spatial domain $\mathbb{Z}_p^m$ whereas $\hat{x}$ denotes the

corresponding vector in the frequency domain $\mathbb{Z}_p^m$.

$$\hat{x}_u = \text{DFT}_{1\text{D}}(x) = \sum_{i=0}^{m-1} x_i \alpha^{iu} \tag{5.2}$$
$$x_i = \text{DFT}_{1\text{D}}^{-1}(\hat{x}) = \frac{1}{m} \sum_{u=0}^{m-1} \hat{x}_u \alpha^{-iu}$$

Note that these transformations can be done locally in MPC, as α and m are public information.

5.1.1 Extending 1D to 2D

The definition in (5.2) can be extended to 2D transformations by transforming all columns first after which all rows are transformed. The matrix X denotes a matrix in the spatial domain $\mathbb{Z}_p^{m\times m}$ and $\hat{X}$ denotes the corresponding matrix in the frequency domain $\mathbb{Z}_p^{m\times m}$. The intermediate result in which all columns are transformed to the frequency domain is called X'. The overall transformation is:

$$\begin{aligned}
\hat{X}_{u,v} &= \text{DFT}_{2\text{D}}(X) = \sum_{j=0}^{m-1}\sum_{i=0}^{m-1} X_{i,j}\alpha^{iu+jv} \\
&= \sum_{j=0}^{m-1} \alpha^{jv}\Big(\sum_{i=0}^{m-1} X_{i,j}\alpha^{iu}\Big) = \sum_{j=0}^{m-1} X'_{u,j}\alpha^{jv} \\
X_{i,j} &= \text{DFT}_{2D}^{-1}(X) = \frac{1}{m^2}\sum_{u=0}^{m-1}\sum_{v=0}^{m-1} \hat{X}_{u,v}\alpha^{-iu-jv} \\
&= \frac{1}{m}\sum_{u=0}^{m-1} \alpha^{-iu}\Big(\frac{1}{m}\sum_{v=0}^{m-1} \hat{X}_{u,v}\alpha^{-jv}\Big) = \frac{1}{m}\sum_{u=0}^{m-1} X'_{u,j}\alpha^{-iu}
\end{aligned} \tag{5.3}$$

Just like (5.2), these transformations can be done locally in MPC because α and m are public.

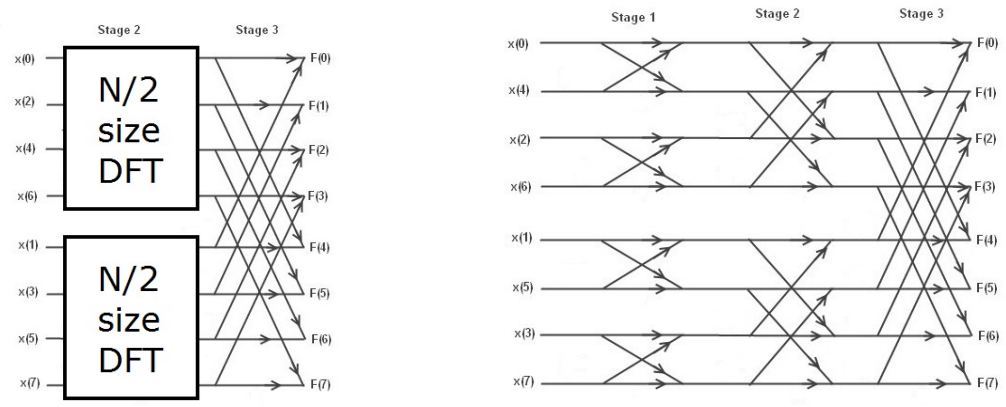
5.1.2 Padding

Definition (2.2) as used in the CNN context is about applying a filter on a matrix. Below the definition it states that if the index of the matrix is out of bounds that we consider the entry to be zero. The definitions in (5.2) and (5.3) however, behave differently. The Fourier transform treats the matrix X as a cyclical object: the column which is left to the leftmost column is considered to be the rightmost column, the row which is above the top row is considered to be the bottom row and so on. This might induce strange behaviour as the convolution might see things that are not there on the edges of the matrices. These false observations might propagate to the middle of the matrices throughout the layers. To prevent these undesired effects, padding is added to the matrices. If the original filter was $\mathbb{Z}_p^{s\times s}$, it suffices to add $\frac{s-1}{2}$ columns with 0s on both sides and add $\frac{s-1}{2}$ rows with 0s on both top and bottom. So if A is in $\mathbb{Z}_p^{n\times n}$ without padding, it is in $\mathbb{Z}_p^{m\times m}$ after padding, with $m = n + s - 1$.

5.2 Comparison with Inner Products

The alternative to using the Fourier transform is evaluating (2.2) as inner products. For this approach it is important to recall that an inner product requires the same amount of communication as an inner product. Applying one filter of size $s \times s$ on a matrix of size $n \times n$ costs n^2 reshares, where the Fourier transform requires m^2 reshares, with $m = n + s - 1$.

To improve the speed of the discrete Fourier transform it is common to apply fast Fourier transform (FFT). It is a technique to evaluate (5.2) for all indices u in fewer multiplications. The main idea is to split the problem into multiple subproblems of equal size, hence the technique works best for powers of 2 or other highly composite numbers. Applying the technique recursively reduces the cost of evaluating (5.2) for all values u from N^2 multiplications down to $O(N\log(N))$. More specifically if N is a power of 2, the FFT costs $N\log_2(N)$ multiplications. The FFT is visualized below for the case $N = 8$:



A graphical display of an (inverse) FFT of array in $\mathbb{R}^N$ with $N = 8$. This image is referred to as the butterfly diagram. Breaking the problem into smaller subproblems, as can be seen on the left, results in the image on the right. The number of diagonal lines equals the number of multiplications that have to be done: $N\log_2(N)$. The diagram illustrates that every input entry has influence on every output.

Assuming m is a power of 2 so we can use the schoolbook 2-radix butterfly method. Then one run of 1D DFT only costs $m\log_2(m)$ runs, such that the full 2D DFT costs $2m$ times as much. Such that the overall complexity of both approaches is based on the following numbers:

$m = n + s - 1$	Fourier in MPC	using inner products
local multiplications	$2m^2\log_2(m)$	s^2n^2
reshares	m^2	n^2
rounds	1	1

Overall the Fourier transform is much less attractive in MPC compared to the clear. No implementation of the FFT has been made in VIFF to compare the calculation times. The only situation in which the FFT would win from the use of inner products is if s would grow to a large number, such that the local multiplications plays a significant role.

However, there are some reasons to believe that s is not likely to grow. Often the designers of a CNN do not give arguments to their specific choice of filter sizes and it is hard to find consensus online on what is good or bad. Some people even claim that designing a good neural network is more an art than a science. An observation that can be made, is that in the ILSVRC papers [5, 9, 10, 11], every network starts with a 7×7 convolutional layer after which there are no kernels bigger than 3×3. With every paper presenting a CNN of at least 50 layers, this means that the 3×3 kernels are vastly popular. On top of that the following following section gives an argument why kernel sizes are not likely to grow. This brings us to the conclusion that Fourier transforms are not efficient in MPC.

5.2.1 Factorizing Convolutions

In a Google paper [18] is explained why it is popular to use small kernels for CNNs. If there is any layer with a large size kernel, the overall costs can be reduced by replacing this layer by multiple convolution layers with smaller kernels. This seems to be the popular belief for CNNs. For example, the computational complexity decreases by replacing a layer with a 5×5 kernel with two layers with 3×3 kernels. Note that the number of parameters in a network has influence on the speed at which the network converges to a working model during the training phase. Taking the number of parameters as a measure a convolution layer with a 5×5 kernel is approximately $\frac{25}{9}$ times as expensive as a convolution layer with a 3×3 kernel. This makes it beneficial to replace one convolution layer with a large size kernel by two convolution layers with a small size kernel, which is called "factorizing convolutions". The idea is that a 5×5 kernel convolution can capture dependencies between signals which are further away from each other in the data compared to a 3×3 kernel convolution. When using two 3×3 kernel convolutions instead, every element in the matrix has information about a 5×5 area in the matrix of 2 layers ago.

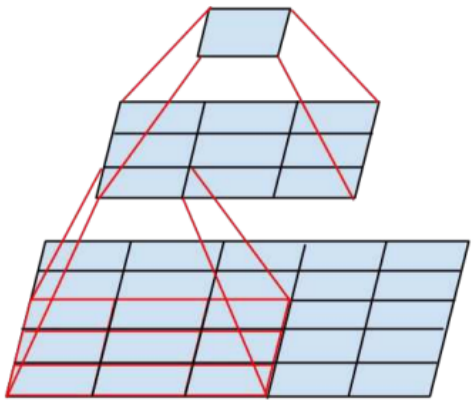


Figure 5.1: How two layers of 3×3 kernel convolutions capture dependencies in a 5×5 grid.

So using two layers with 3×3 kernels is believed to be more efficient in terms of parameter usage.

Chapter 6

New Protocols

In this section new protocols are proposed for the LSB, truncation and comparison. The improvements for the LSB, the truncation and the pseudo random bit are implemented for (t,n) with $t = 1$ and are faster/better than the current implementations. The improved protocol for the comparison is implemented for $t = 1$ and shows improved calculation time. Even more improvement can be made when it is combined with the new pseudo random bit protocol for $t = 1$. The comparison has also been improved for $t = 2$. No implementation has been made for $t = 2$, but the theoretical analysis of the complexity improvement looks promising.

6.1 Cheap Truncate for $t = 1$

In a paper about SecureML [15] a setup is mentioned where additive secret sharing is used. In this setting all parties perform a local truncation on their share of a secret. The resulting shares can be recombined to a number which is very close to the truncated value of the original secret. The overall multi-party truncation is performed with only a small error and most of the non linear computations are done locally. This approach can also be applied to Shamir secret sharing. When a qualified subset agrees to perform the new protocol, every player of the qualified subset applies a linear operator to their share of a secret. This way an additive secret sharing can be obtained in which the addition must be done with the same modulus as the Shamir secret sharing. The rest of the protocol is similar to the mentioned protocol in the SecureML paper [15]. The modulus is a prime number which gives us a restriction on the number of players. The new protocol is only faster than the previous one if the qualified subset is of size $2 = t + 1$, which is the case in an honest majority 3 party computation. The 3 party honest majority setting is the smallest honest majority setting to profit from the benefits of Shamir secret sharing. The local computional complexity of PRSS as mentioned in Section 3.3 is also superexponential in terms of n, which makes the use of small numbers n more attractive. The suggested protocol is only implemented for 3 parties in VIFF, but the performance should be equal in any (t,n)-secret sharing scheme with $t = 1$. The 3 party implementation of the probabilistic truncation is compared to the old protocol for probabilistic truncations which can be read in the PhD thesis of Sebastiaan de Hoogh [6]. The new protocol is implemented in VIFF and shows improvements in the calculation time over the old protocol. The new protocol can be adjusted for $t > 1$, but the protocol of de Hoogh has a lower round complexity for $t > 1$.

6.1.1 Protocol for Cheap Probabilistic Truncate

In additive secret sharing where the modulus is a big power of 2 every player can round his secret share down to a multiple of 2^f, after which the sum of the secret shares is a multiple of 2^f. This is the core argument to the probabilistic truncation protocol proposed in this section. This method could be used to approximate any floor division with an integer divisor. Because our application of the following protocol is the reduction of significant bits, the focus is on floor division with the divisor being $2^f < p, f \in \mathbb{N}$. The protocol could be applicable in a wide range of numerical applications in Shamir secret sharing. The mathematical principles behind the protocol will be explained after showing the pseudo code. Note that // denotes integer floor division.

Protocol 9 [c] $\leftarrow$ NewProbTrunc$_{t=1}([x], f)$

Require:
$2^f << p$

1: **for each party** i=1,2 **do**
2: $\quad s_i \leftarrow [x + 2^{k-1}]_i \prod_{j=1, j\neq i}^{2} \frac{-j}{i-j} \bmod p$
3: $\quad q_i \leftarrow [(s_i - (1-i)p)//2^f] \cdot 2^f \bmod p$
4: $\quad [q_i] \leftarrow \text{SShare}(q_i, 1, 3)$ 1 rnd, 1 inv
5: $[c] \leftarrow ([q_1] + [q_2]) \cdot 2^{-f} - 2^{k-1-f}$
6: **Return** $[c]$

The protocol costs 1 rounds and 1 invocation, which is cheaper than the probabilistic truncate described by de Hoogh [6] which costs 2 rounds and $f+1$ invocations. In terms of local computations the new protocol also has an advantage over the protocol of de Hoogh, as it requires no exponentiations. In the old protocol these exponentiations were required for computing random bits. The following runtimes are from experimental runs performing 15.000 truncations on a HP Elitebook 8540w with different divisors 2^f and protocols:

	Protocol 11	old truncation protocol
$f = 2$	11.03s	35.77s
$f = 4$	11.77s	39.39s
$f = 8$	10.76s	41.73s
$f = 16$	10.56s	52.4s

As discussed in Section 4.2.1, neural network are expected to be evaluable with 8 bits. From this fact we can conclude that in our application the implementation of Protocol 9 gives an approximate factor 4 of speed improvement.

6.1.2 Theory Behind Protocol

In line 2 of Protocol 9 additive shares are constructed such that $s_1 + s_2 \equiv x + 2^{k-1} \bmod p$. The coefficients to construct these numbers come from the Lagrange interpolation in (3.1). For example if $D = \{1,2\}$, then $s_1 = 2([x]_1 + 2^{k-1})$ and $s_2 = -1([x]_2 + 2^{k-1})$. The goal is now to round down these s_i to q_i which means

$$q_i \equiv s_i - \Delta_i \bmod p,\ \Delta_i \in [0, 2^f) \tag{6.1}$$

Here it can be seen that q_i is constructed from s_i using only a subtraction, which is known to be cheap in Shamir secret sharing. The constructed q_i satisfies the following

$$q_1 + q_2 \equiv \tilde{x} \bmod p \tag{6.2}$$
$$\tilde{x} \in [0, p) \tag{6.3}$$
$$\tilde{x} \equiv 0 \bmod 2^f \tag{6.4}$$

To meet the conditions in conditions in (6.1),(6.2) and (6.4) it is important to distinguish two cases. Notation: let $(a \bmod p)$ denote the integer $b \in [0, p)$ such that $b \equiv a \bmod p$.

$$\text{case 1} : (q_1 \bmod p) + (q_2 \bmod p) = p + \tilde{x} \tag{6.5}$$
$$\text{case 2} : (q_1 \bmod p) + (q_2 \bmod p) = \tilde{x} \tag{6.6}$$

As will be proven in this section, case 2 happens only with negligible probability. Therefore Protocol 9 is built around case 1.

Theorem 6.1.1. *If both*

- $t = 1$ *i.e.* $[x] = x + i \cdot r$ *for some* $r \in \mathbb{Z}$
- $x \in [0, p/2^\kappa]$ *and* $q_i \equiv \frac{-j}{i-j}[x]_i \bmod p$

then $(q_1 \bmod p) + (q_2 \bmod p) \geq p$ *happens with a probability not bigger than* $1 - 2^{-\kappa}$.

proof

If $(q_1 \bmod p) > \tilde{x}$ holds, then (6.6) cannot be the case as $(q_2 \bmod p) \geq 0$, which implies $(q_1 \bmod p) + (q_2 \bmod p) > \tilde{x}$, so (6.6) cannot hold.
Hence $(q_1 \bmod p) > \tilde{x} \Rightarrow (q_1 \bmod p) + (q_2 \bmod p) \overset{6.5}{=} p + \tilde{x}$

As $q_1 \in_R \mathbb{Z}_p$ and $p > 2^{k+\kappa}$ for some security parameter κ, it follows that

$$\mathbb{P}((q_1 \bmod p) > \tilde{x}) \geq \mathbb{P}((q_1 \bmod p) > 2^k) \geq 1 - 2^{-\kappa}$$

Which finishes the proof of Theorem 6.1.1 □

As Theorem 6.1.1 tells that $\mathbb{P}((q_1 \bmod p) \leq \tilde{x}) < 2^{-\kappa-1}$ is negligible, it can be assumed that (6.5) is always the case, in which case the following definition gives a deterministic way of finding good values for q_i

$$q_1 := (((s_1 \bmod p) - p)//2^f) \cdot 2^f$$
$$q_2 := ((s_2 \bmod p)//2^f) \cdot 2^f$$

The conditions in (6.1),(6.2) and (6.4) are met by the q_i that are produced in Protol 9. Protocol 9 produces (q_1, q_2) deterministicly for every pair of shares $([x]_1, [x]_2)$. But since there are multiple pairs of shares representing the same x, this protocol does not result in a unique solution $\tilde{x}$ for a given x. So even within case 1 there are two possible values $\tilde{x}$ that can occur as an output: either $[(x//2^f) \cdot 2^f]$ or $[(x//2^f - 1) \cdot 2^f]$. If both Δ_1 and Δ_2 are big in (6.1) then $\tilde{x} = [(x//2^f - 1) \cdot 2^f]$ is more likely to be the case. More precisely

$$\Delta_1 + \Delta_2 \geq 2^f \Leftrightarrow \tilde{x} = (x//2^f - 1) \cdot 2^f \tag{6.7}$$

Both Δ_i values are known to one party, but guessing whether $\Delta_1 + \Delta_2 \geq 2^f$ is the case is as hard as computing a comparison.

Relation between Truncation and Comparison

Any truncation protocol which gives the correct answer all the time, can be used to construct a comparison operation in the following way:

Protocol 10 [c] $\leftarrow$ LessThanZero($[x], f$)

Require:

$x \in [-2^f, 2^f)$

1: $[b] \leftarrow -\text{Trunc}([x], f)$
2: **Return** $[b]$

In which b is the comparison bit $x < 0$. Protocol 10 is at least as easy as performing a Trunc([x], f) in terms of communication complexity. The cheapest comparison operation described by de Hoogh[6] costs $O(\log(f))$ rounds and $O(f)$ invocations. Assuming that this is optimal it is reasonable that Protocol 9 is not giving the correct output all the time.

6.1.3 Output Probability Distribution

The output of protocol 9 is either $[x//2^f]$ or $[x//2^f - 1]$. As can be seen by (6.7), we have a good answer when

$$\Delta_1 + \Delta_2 < 2^f \Leftrightarrow q_1 + q_2 \equiv x - (x \bmod 2^f) \bmod p \Leftrightarrow [c] = [x//2^f]$$

And the other case returns $[c] = [x//2^f - 1]$. So protocol 9 gives the correct output or it returns the correct answer minus 1.

The probability of producing the right answer $[x//2^f]$ depends mainly on $(x \bmod 2^f)$. To give an expression for the probability distribution it is convenient to approximate $(s_1 \bmod 2^f)$ by a random variable $Y \in_R [0, 2^f)$. The following calculation shows the statistical distance between the two random variables:

$$X := \{0, 1, \ldots, 2^f - 1\} \tag{6.8}$$

$$S := (s_1 \bmod 2^f) \tag{6.9}$$

$$\forall x \in X : \ \mathbb{P}(S = x) = \begin{cases} \frac{\lceil \frac{p}{2^f} \rceil}{p} \text{ if x} < (\text{p mod } 2^{\text{f}}) \\ \frac{\lfloor \frac{p}{2^f} \rfloor}{p} \text{ else} \end{cases} \tag{6.10}$$

$$\forall x \in X : \ \mathbb{P}(Y = x) = \frac{1}{2^f} \tag{6.11}$$

$$X^+ = \{x \in X | \mathbb{P}(S = x) > \mathbb{P}(Y = x)\} \tag{6.12}$$

$$= \{0, 1, \ldots, (p \bmod 2^f) - 1\}$$

$$\Delta(S, Y) = \sum_{x \in X^+} (\mathbb{P}(S = x) - \mathbb{P}(Y = x)) \tag{6.13}$$

$$= (p \bmod 2^f)(\frac{\lceil \frac{p}{2^f} \rceil}{p} - \frac{1}{2^f}) = (p \bmod 2^f)(\frac{\lceil \frac{p}{2^f} \rceil 2^f - p}{2^f p})$$

$$= (p \bmod 2^f)(\frac{2^f - (p \bmod 2^f)}{2^f p}) \leq 2^{f-1} \frac{2^{f-1}}{2^f p} = \frac{2^{f-2}}{p} < 2^{-\kappa - 2}$$

With a negligible statistical distance between S and Y, the distribution of $Y \in_R X$ can be used to express $\mathbb{P}([c] = [x//2^f])$:

$$\mathbb{P}([c] = [x//2^f]) = \mathbb{P}(\Delta_1 + \Delta_2 < 2^f) = \mathbb{P}(\Delta_1 \leq (x \bmod 2^f)) = \frac{(x \bmod 2^f)}{2^f} \tag{6.14}$$

If $(x \bmod 2^f)$ would also be considered as a uniform random variable, then the probability of a correct output of Protocol 9 is

$$\mathbb{P}([c] = [x//2^f]) = \frac{2^f + 1}{2 \cdot 2^f} \approx \frac{1}{2}$$

6.1.4 Extension to more Parties

For more parties Theorem 6.1.1 can be altered slightly to prove that $\sum_{i=1}^{t+1}(q_i \bmod p) \geq p$ with overwhelming probability. However with more parties, more cases like (6.5) and (6.6) need to be distinguished as the total number of overflows needs to be known. To find the number of overflows LSBs can be used. For example for $(t,n) = (2,5)$, the number of overflows can be measured with the least significant bit of $\tilde{x}$ in the following way:

Protocol 11 [c] $\leftarrow$ NewProbTrunc$_{t=2}([x], f)$

1: **for party** i=1 **do**
2: $\quad s_i \leftarrow [x + 2^{k-1}]_i \prod_{j=1, j\neq i}^{2} \frac{-j}{i-j} \bmod p$
3: $\quad s_i \leftarrow s_i - p$
4: **for each party** i=2,3 **do**
5: $\quad s_i \leftarrow [x + 2^{k-1}]_i \prod_{j=1, j\neq i}^{2} \frac{-j}{i-j} \bmod p$
6: **for each party** i=1,2,3 **do**
7: $\quad q_i \leftarrow (s_i//2^f)2^f \bmod p$
8: $\quad [q_i] \leftarrow \text{SShare}(q_i, 1, 3)$ — 1 rnd, 1 inv
9: $[b] \leftarrow \text{LSB}([q_1] + [q_2] + [q_3])$ — 2rnd, 2 inv
10: $[c'] \leftarrow ([q_1] + [q_2] + [q_3] + (-p \bmod 2^f)[b])2^{-f} - 2^{k-1-f} \bmod p$
11: **Return** $[c']$

A similar approach would be possible with even more parties. Line 9 could then be replaced by a for loop in which the $\lceil \log_2(t) \rceil$ least significant bits need to be reconstructed to find the exact number of overflows. This makes the overall complexity $1 + 2\lceil \log_2(t) \rceil$ rounds and $1 + 2\lceil \log_2(t) \rceil$ invocations for $t > 1$. The for loop is not the most efficient way to perform this protocol in terms of round complexity, but the main takeaway is that this protocol is especially efficient for (t,n) with $t = 1$

In cases with $t > 1$, it is less obvious which approach is better for the probabilistic truncate, but here are the known characteristics of both protocols:

	Probabilistic truncation from de Hoogh [6]	Protocol 11
local complexity	exponentiations for random bits	no exponentiations
rounds	2	$1 + 2\lceil \log_2(t) \rceil$
invocations	f+1	$1 + 2\lceil \log_2(t) \rceil$
output range	$\{x//2^f + \delta \mid \delta \in \{0,1\}\}$	$\{x//2^f + \delta \mid \delta \in \{-t, \ldots, 0\}\}$

6.2 LSB for $t = 1$

Theorem 6.1.1 can also be used to calculate least significant bits. When the additive shares s_i are calculated the parity of s_i can be calculated locally. De Hoogh [6] also presents an approach for taking LSBs, in which he also obtains two bits which are to be XORed to find the actual LSB. The bits $b_i := (s_i \bmod 2)$ perform a similar role. Individually b_1 and b_2 do not leak any information on x, but $[b_1] \oplus [b_2] = [b_1] + [b_2] - 2 \cdot [b_1] \cdot [b_2] = [\mathrm{LSB(x)}]$. So the following approach first creates the bits b_i after which it will XOR the bits securely:

Protocol 12 $[\mathrm{c}] \leftarrow \mathrm{LSB}_{t=1}([x])$

1: **for each party** i=1,2 **do**
2: $\quad s_i \leftarrow [x + 2^{k-1}]_i \prod_{j=1, j\neq i}^{2} \frac{-j}{i-j} \bmod p$
3: $\quad b_i \leftarrow ([(s_i - (1-i)p)] \bmod 2)$
4: $\quad [b_i] \leftarrow \mathrm{SShare}(b_i, 1, 3)$ — 1 rnd, 1 inv
5: $[c] \leftarrow [b_1] + [b_2] - 2[b_1][b_2]$ — 1 rnd, 1 inv
6: **Return** $[c]$

Protocol 12 costs 2 rounds and 2 invocations, just as the LSB protocol described by de Hoogh [6]. As this new approach requires less local computation, the implementation of the new protocol on a HP Elitebook 8540w is 30% faster than VIFFs current LSB as described by de Hoogh [6]. In this experiment 50.000 LSBs are calculated in a $(t,n) = (1,3)$ VIFF implementation, which takes 1 minute and 14 seconds with Protocol12 and 1 minute and 46 seconds with the old protocol.

6.2.1 New LSB Protocol for $t > 1$

Protocol 12 works for any (t,n) threshold scheme with $t = 1$. If a similar approach would be written for $t > 1$, then more bits b_i need to be constructed. Then two problems would arise:

- Theorem 6.1.1 does not provide us with the exact number of overflows. To find out the exact number of overflows that occur, the protocol could take $\lceil \log_2(t) \rceil$ LSBs from the sum of rounded down q_i values, similar to Protocol 11 line 9. Just like Protocol 11 this would result in a higher communication complexity compared to the $t = 1$ case.
- There are more bits to xor which results in more communication costs. When the XORs are calculated with divide and conquer, the overall XOR process costs $\lceil \log_2(t+1) \rceil$ rounds and t invocations.

This leads to an overall communication complexity of $1 + 2\lceil \log_2(t) \rceil + \lceil \log_2(t) \rceil$ rounds and $1 + 2\lceil \log_2(t) \rceil + t$ invocations. For $t > 1$ this is strictly worse than the 2 rounds and 2 invocations, which are the costs of the LSB described by de Hoogh [6].

6.3 Comparison for $t = 1$ and $t = 2$

The current comparison that is implemented in VIFF has a round complexity of $1 + \lceil \log_2(\ell + 2) \rceil$. In this section we suggest an enhancement to this protocol which results in 3 rounds for $t = 1$ and 4 rounds for $t = 2$. Unfortunately we do not see how to extend the new protocol to $t > 2$ and only found time to implement it for $t = 1$.

6.3.1 the Comparison Protocol in VIFF

In Protocol 8 a sketch is given to show that a comparison requires more communication than a multiplication in MPC. The actual comparison operation which is implemented in VIFF is more complex, more efficient and harder to understand than Protocol 8. Therefore the rough idea of the protocol will be explained before showing the pseudocode. As can be read in [8], the logarithmic round comparison is based on the following principle.

Let $x = X_1X_0$ and $y = Y_1Y_0$ be bit strings with $|X_1| = |Y_1|$ and $|X_0| = |Y_0|$. Let $x > y$ denote the bit from $\{0,1\}$ that corresponds to the boolean $x > y$.

$$[x > y] = \begin{cases} [X_1 > Y_1], & X_1 \neq Y_1 \\ [X_0 > Y_0], & X_1 = Y_1 \end{cases}$$

This property can be exploited in the formula

$$[x > y] = [X_1 > Y_1] + [X_1 = Y_1][X_0 > Y_0]$$

The above property can be applied multiple times by breaking the strings into smaller strings until the point that x and y are compared bit by bit. This property is used in the following protocol. The resulting formula shows that the answer $x > y$ can be found by comparing bits from left to right. The first bit $x_i \neq y_i$ determines which number is bigger. The comparison protocol compares the bits from left to right and for the purpose of bookkeeping an $[X]$ is introduced. As long as all observed bits are pairwise equal ($x_i == y_i$) the $[X]$ is 0. As soon as $[X] \neq 0$, the bit comparisons do not matter anymore, but no party will notice that $[X] \neq 0$ as the secret $[X]$ is never opened. The properties from this $[X]$ is used to construct a sequence $([e_i])$ which can be constructed locally with only linear operations. This sequence is constructed in such a way that the eventual question is: "Is there an i such that $e_i = 0$?".

If the j-th bit is the first bit such that $x_j \neq y_j$, then e_j is the element that is possibly 0. But publishing this j itself would leak information about x and y. Hence this question is answered by only opening the value $r \cdot \prod_i e_i$ in which $r \in_R \mathbb{Z}_p$ to mask the actual product if it is not 0.

Because it is cheaper to answer this question by opening a share, the comparison is reversed (before the construction of $([e_i])$) with probability $\frac{1}{2}$. So the question that is being answered in the clear is $x \leq y$ or $x > y$, but nobody knows which one. For this reason no information is leaked by publicly telling whether there was an i such that $e_i = 0$.

The pseudo code of the logarithmic round less than zero comparison is the following:

Protocol 13 $[b] \leftarrow \text{LTZ}_{t=1}([x], \ell)$ returning $[x \leq 0]$

Require:
$x \in [-2^\ell, 2^\ell)$

1: **for each** $i = 0$ **to** $i = \ell - 1$
2: $\quad [r_i] \leftarrow \text{PRandBit}$ 1rnd, ℓ inv
3: $[R] \leftarrow \text{PRandInt}([0, 2^{\kappa+k-\ell}])$ κ security parameter, k max bit length secret
4: $[c] \leftarrow [x] + \sum_{i=0}^{\ell-1} [r_i] 2^i + 2^\ell \cdot [R]$
5: $c \leftarrow (SOpen([c]) \bmod 2^\ell)$
6: $[s_{\text{bit}}] \leftarrow \text{PRandBit}$ 1 inv
7: $[s] = 2 \cdot [s_{\text{bit}}] - 1$
8: $[X] \leftarrow 0$
9: **for** $i = 1$ **to** $\ell - 1$
10: $\quad c_i \leftarrow ((c//2^i) \bmod 2)$
11: $\quad [e_i] \leftarrow [s] + [r_i] + c_i + 3 \cdot [X]$
12: $\quad [X] \leftarrow [X] + c_i + (1 - 2c_i)[r_i]$ $[X] += [r_i] \oplus c_i$
13: $[e_\ell] = 1 - [s_{\text{bit}}] + 3[X]$
14: $[e_{\ell+1}] \leftarrow \text{PRandField}$ mask product in case $e_i \neq 0 \; \forall i$
15: $[f] = \prod_{i=0}^{\ell+1} [e_i]$ $\log_2(\ell+2)$ rnd, $\ell + 1$ inv
16: $f \leftarrow \text{SOpen}([f])$
17: **if** $f = 0$:
18: $\quad [b] \leftarrow ([x] + \sum_{i=0}^{\ell-1} [r_i] 2^i - (c + [s_{\text{bit}}] \cdot 2^\ell))/2^\ell$
19: **else**
20: $\quad [b] \leftarrow ([x] + \sum_{i=0}^{\ell-1} [r_i] 2^i - (c + (1 - [s_{\text{bit}}]) \cdot 2^\ell))/2^\ell$
21: **return** $[b]$

The complexity of this algorithm is $1 + \lceil \log_2(\ell+2) \rceil$ rounds and $2\ell + 2$ reshares. The main cause of this round complexity can be found in line 15 of Protocol 13. As a reshare is needed after every multiplication, the optimal number of rounds can be achieve by using a divide and conquer strategy, which results in the $\lceil log_2(\ell+2) \rceil$ rounds for line 15 alone.

6.3.2 The Improvement on the Comparison

Instead of using a masked product of $\prod_i e_i$, the new protocol investigates the question "Is there an i such that $e_i = 0$?" by using a zero test. In the $t = 1$ player case we need $t + 1 = 2$ players to construct the sequence $([e_i])$ as done before. The players are able to generate random mask numbers t_i and s_i for every $[e_i]$ without communication using PRSS. If every player $j = 0, 1$ sends the value $t_i[e_i] + (-1)^j s_i$ to player 2, he can combine them in such a way to find the value $t_i \cdot e_i$. Note that $t_i = 0$ with negligible probability, so player 2 sees only whether an e_i is 0, as all other values are masked by the t_i. To prevent that player 2 finds out which e_i is 0, players 0 and 1 rotate the sequence e_i before sending.

The benefit of this approach is that the zero tests can run in parallel.

This results in the following pseudo code:

Protocol 14 $[b] \leftarrow \text{NewLTZ}_{t=1}([x], \ell)$ returning $[x \leq 0]$

Require:
$x \in [-2^\ell, 2^\ell)$

for each $i = 0$ **to** $i = \ell - 1$
  $[r_i] \leftarrow$ PRandBit — 1rnd, ℓ inv
$[R] \leftarrow \text{PRandInt}([0, 2^{\kappa+k-\ell}])$ — κ security parameter, k max bit length secret
$[c] \leftarrow [x] + \sum_{i=0}^{\ell-1} [r_i] 2^i + 2^\ell \cdot [R]$
$c \leftarrow SOpen([c])$
$[s_{\text{bit}}] \leftarrow$ PRandBit — 1 inv
$[s] = 2 \cdot [s_{\text{bit}}] - 1$
$[X] \leftarrow 0$
for party $j = 0, 1$ **do**
  for $i = \ell - 1$ **to** 0 — from most to least significant bit
    $c_i \leftarrow ((c//2^i) \bmod 2)$
    $[e_i] \leftarrow [s] + [r_i] + c_i + 3 \cdot [X]$
    $[X] \leftarrow [X] + c_i + (1 - 2c_i)[r_i]$ — $[X] += [r_i] \oplus c_i$
  $[e_l] = 1 - [s_{\text{bit}}] + 3[X]$
  $v \leftarrow$ PRandField — rotation to mask which $e_i == 0$
  for $i = 0$ **to** ℓ
    $t_i \leftarrow$ PRandField
    $s_i \leftarrow$ PRandField
    $f_{i,j} \leftarrow t_i [e_{(i+v \bmod \ell+1)}]_j + (-1)^j \cdot s_i$
    send $f_{i,j}$ **to party** 2 — 1 rnd, ℓ inv
for party 2 **do**
  for $i = 0$ **to** ℓ
    $f_i \leftarrow 2 \cdot f_{i,0} - f_{i,1}$
  if $\exists i : f_i == 0$:
    send $f = 0$ **to party** $0, 1$
  else
    send $f = 1$ **to party** $0, 1$
if $f = 0$:
  $[b] \leftarrow ([x] + \sum_{i=0}^{\ell-1} [r_i] 2^i - c)/2^\ell + [s_{\text{bit}}]$
else
  $[b] \leftarrow ([x] + \sum_{i=0}^{\ell-1} [r_i] 2^i - c)/2^\ell + 1 - [s_{\text{bit}}]$
return $[b]$

The complexity of Protocol is 3 rounds, $2\ell + 3$ reshares

6.3.3 Experimental Results

Protocol 14 has an asymmetrical role division for the participating parties. Let us call the two parties that generate $f_{i,j}$ elements the constructors and the party that receives the $f_{i,j}$ elements the opener. Protocol 14 has been implemented together with "rotated Protocol 14" versions. In these rotated versions of Protocol 14 every party is the opener in $\frac{1}{3}$ of the comparisons. If the workload is not equally divided over the openers and the constructors this rotation divides the workload better, also different communication channels are used in the rotated versions of the protocol.

$\ell = 32$	Protocol 13	Protocol 14	rotated Protocol 14
calculation time (s)	13.9	8.6	8.4

The rotated version of the protocol comes with more bookkeeping and is not considered as a significant improvement over Protocol 14 and is therefore considered not worth the effort. For the sake of comparison it might be more relevant to notice that all these protocols start with taking random bits r_i and a s_{bit}. If the process of taking this random bits is repeated for a 1000 times in parallel, the calculation time is 5.9 seconds on average. This implies that the second half (line 9 and further) of the comparison protocol has been improved from $13.9 - 5.9 = 8$ to $8.6 - 5.9 = 2.7$ seconds. This difference only becomes larger as ℓ grows.

6.3.4 Protocol for $t = 2$

Protocol 14 can be written for $t > 1$. The group of constructors would be $t + 1$ parties instead of 2 and the opener would still be able to tell whether there is a 0. The opener is still not able to learn something about x on his own. But a (t,n) secret sharing scheme should prevent any subset of t or less parties to learn anything about the secrets. When the opener cooperates with any constructor they would learn which $e_i == 0$ because the constructor knows v. Knowing which $e_i == 0$ leaks information on x.

To prevent any set of 2 players to learn information about x an extra role is introduced. The system has constructors, a "combiner" and a "tester". The rough layout of the protocol would be the following: The constructors do the same thing as in the $t = 1$ case constructing $f_{i,j}$ elements, but instead of sending $f_{i,j}$ to the opener, they send $f_{i,j} + w_j$ to the combiner and sends w_j to the opener. The main idea of adding w_j is that this value is only known to this constructor and the combiner. This way the zero test is replaced by a $= \sum_j \lambda_j \cdot w_j$ test which can only be performed by a party that knows all values w_j.

$$\sum_j \lambda_j \cdot f_{i,j} == 0 \Leftrightarrow \sum_j \lambda_j \cdot (f_{i,j} + w_{i,j}) == \sum \lambda_j w_j$$

Hence the combiner calculates $\sum_j \lambda_j \cdot (f_{i,j} + w_j)$ which results in the sequence f_i. The combiner rotates this sequence f_i and sends this $\tilde{f}_i$ to the opener. The opener tests $\tilde{f}_i == \sum_j \lambda_j \cdot w_j$ and reports this to everybody as answer to the question "Is there an i such that $e_i == 0$?".

If an evil subset of size would try to learn anything about x in a similar fashion as before, they must know the rotation from the constructors, the rotation of the combiner and the index of $\tilde{f}_i == 0$. This means this evil subset would at least need 3 parties and any subset of 3 parties is a qualified subset so this is not a weakness of the protocol.

Protocol 15 $[b] \leftarrow \text{NewLTZ}_{t=2}([x], \ell)$ returning $[x \leq 0]$

1: **for each** $i = 0$ **to** $i = \ell - 1$
2: $\quad [r_i] \leftarrow \text{PRandBit}$ 1rnd, ℓ inv
3: $[R] \leftarrow \text{PRandInt}([0, 2^\kappa])$ κ security parameter
4: $[c] \leftarrow [x] + \sum_{i=0}^{\ell-1} [r_i] 2^i + 2^\ell \cdot R$
5: $c \leftarrow SOpen([c])$
6: $[s_{\text{bit}}] \leftarrow \text{PRandBit}$ 1 inv
7: $[s] = 2 \cdot [s_{\text{bit}}] - 1$
8: $[X] \leftarrow 0$
9: **for party** $j = 0, 1, 2$ **do**
10: $\quad$ **for** $i = \ell - 1$ **to** 0 from most to least significant bit
11: $\quad\quad c_i \leftarrow ((c//2^i) \bmod 2)$
12: $\quad\quad [e_i] \leftarrow [s] + [r_i] + c_i + 3 \cdot [X]$
13: $\quad\quad [X] \leftarrow [X] + c_i + (1 - 2c_i)[r_i]$ $[X] += [r_i] \oplus c_i$
14: $\quad [e_l] = 1 - [s_{\text{bit}}] + 3[X]$
15: $\quad v \leftarrow \text{PRandField}$ PRSS: known among parties $0, 1, 2$
16: $\quad w_j \leftarrow_R \mathbb{Z}_p$ only known to party j
17: $\quad$ **for** $i = 0$ **to** ℓ
18: $\quad\quad t_i \leftarrow \text{PRandField}$ PRSS: known among parties $0, 1, 2$
19: $\quad\quad f_{i,j} \leftarrow t_i [e_{(i+v \bmod \ell+1)}]_j + w_j$
20: $\quad\quad$ **send** $f_{i,j}$ **to party** 3 1 rnd, ℓ inv
21: **for party** 3 **do**
22: $\quad u \leftarrow_R \{0, 1, \cdot, \ell\}$
23: $\quad$ **for** $i = 0$ **to** ℓ
24: $\quad\quad f_i \leftarrow 3 \cdot f_{i,0} - 3 f_{i,1} + f_{i,2}$ Lagrange interpolation
25: $\quad\quad \tilde{f}_i \leftarrow f_{i+u \bmod \ell+1}$
26: $\quad\quad$ **send** $\tilde{f}_i$ **to party** 4 1 rnd, ℓ inv
27: **for party** 4 **do**
28: $\quad$ **for** $i = 0$ **to** ℓ
29: $\quad$ **if** $\exists i : \tilde{f}_i == 3w_0 - 3w_1 + w_2$:
30: $\quad\quad$ **send** $f = 0$ **to party** $0, 1, 2, 3$
31: $\quad$ **else**
32: $\quad\quad$ **send** $f = 1$ **to party** $0, 1, 2, 3$
33: **if** $f = 0$:
34: $\quad [b] \leftarrow ([x] + \sum_{i=0}^{\ell-1} [r_i] 2^i - c)/2^\ell + [s_{\text{bit}}]$
35: **else**
36: $\quad [b] \leftarrow ([x] + \sum_{i=0}^{\ell-1} [r_i] 2^i - c)/2^\ell + 1 - [s_{\text{bit}}]$
37: **return** $[b]$

The protocol has not been implemented, but the communication complexity of this protocol has only increased by 1 round compared to Protocol 14. The complexity for $t = 2$ is 4 rounds and $3\ell + 4$ reshares.

6.4 Random Bits for $t = 1$

In the very last phase of this study a new method has been discovered to calculate pseudo random bits. The method requires 1 invocation just like Protocol 5, but requires less computational complexity.

Three parties $p1$, $p2$ and $p3$ are involved in the protocol. Every pair of players calculate a random bit using PRSS, which are used to calculate shares $[b]_i$. Parties $p1$ and $p2$ know bit $b2$, parties $p2$ and $p3$ know bit $b3$ and parties $p1$ and $p3$ know bit $b1$. Those bits enable the parties to calculate a 2nd degree polynomial that encodes a pseudo random bit which is unknown to everybody.

Some logical deduction can be applied to the underlying secret of this pseudo random bit. Let us call the underlying secret $b = g(b_1, b_2, b_3) \in \{0,1\}$. If party $p1$ sees that $b1 = b2 = 0$, then the answer might not be known to $p1$ based on these facts and therefore $g(0,0,0) = 1 - g(0,0,1)$. Similar symmetries tell us that the more general rule

$$g(b_1, b_2, b_3) = 1 - g(1 - b_1, b_2, b_3) = 1 - g(b_1, 1 - b_2, b_3) = 1 - g(b_1, b_2, 1 - b_3)$$

must hold. Assuming that $g(0,0,0) = 0$ this gives $g(b_1, b_2, b_3) = b_1 \oplus b_2 \oplus b_3$. The other solution with $g(0,0,0) = 1$ is $g(b_1, b_2, b_3) = 1 - b_1 \oplus b_2 \oplus b_3$.

Every party is able to anticipate on the 2 bits he knows, so 4 cases can be distinguished by every party. Every party i can look up $[b]_i$ in the table below based on the 2 bits he knows. This table works for every value $a, c, d, e \in \mathbb{Z}_p$.

b_1	b_2	b_3	b $b_1 \oplus b_2 \oplus b_3$	$p_1(\lambda_1 = 3)$ knows b_1, b_2 $[b]_1$	$p_2(\lambda_2 = -3)$ knows b_2, b_3 $[b]_2$	$p_3(\lambda_3 = 1)$ knows b_1, b_3 $[b]_3$
0	0	0	0	a	c	$3c - 3a$
0	0	1	1	a	d	$1 + 3d - 3a$
0	1	0	1	$\frac{1}{3} + a - d$	$c - d$	$3c - 3a$
0	1	1	0	$\frac{1}{3}$+a-d	$\frac{-2}{3}$	$1 + 3d - 3a$
1	0	0	1	e	c	$1 + 3c - 3e$
1	0	1	0	e	d	$3d - 3e$
1	1	0	0	$\frac{-1}{3} + e - d$	$c - d$	$1 + 3c - 3e$
1	1	1	1	$\frac{-1}{3} + e - d$	$\frac{-2}{3}$	$3d - 3e$

Table 6.1: The lookup table to construct a 2nd degree polynomial encoding a random bit. This construction requires no communication

After constructing this 2nd degree polynomial, only a reshare is needed to get a 1st degree polynomial which is usable in all kinds of applications.

If $n > 3$ the parties $p4, p5, \ldots$ do not do anything and just wait for the reshare in which they get a share $[b]_i$.

6.4.1 Benefits

The benefits of this approach are

- This approach requires less computational complexity than Protocol 5 and is therefore faster.
- Having the 2nd degree polynomial can already be useful in some applications. For example, in Protocol 6 in the case $(t,n) = (1,3)$, the SOpen would also accept a 2nd degree polynomial for b. This means that the round complexity of Protocol 6 can be reduced to 1 round as the open can be evaluated in parallel to the reduction of b.

6.4.2 Analysis for $t > 1$

This method is not extendable for $t > 1$, which can be seen by counting the number of variables and the number of constraints in the system for $(t, 2t+1)$ parties. For the following analysis we assume $n = 2t+1$. If any subset of t parties collectively knows all bits b_i, then this approach is not considered secure. Hence $\binom{n}{t}$ bits need to be generated. Every bit is unknown to a differend t sized subset of parties.

Then every party knows $\binom{n-1}{t}$ bits on which it can anticipate. If a similar table to Table 6.1 would be constructed every column can have at most $2^{\binom{n-1}{t}}$ different values in the column of p_i and there are n such columns. Therefore there are $2^{\binom{n-1}{t}} \cdot n$ degrees of freedom. The remaining constraints are the Lagrange interpolation constraints that state that every row should have entries such that the values encode $b = \bigoplus_i b_i$. Every row of the table would imply one such constraint and there are $2^{\binom{n}{t}}$ rows as there are $\binom{n}{t}$ bits that are generated with PRSS.

For $(t,n) = (1,3)$ this formula results in 12 variables with 8 constraints, which results in 4 degrees of freedom $a,c,d,e \in \mathbb{Z}_p$. For $(t,n) = (2,5)$ this results in 320 variables with 1024 constraints, which implies that no solution exists.

Chapter 7

Conclusion

The first goal of this thesis was to implement a neural network in MPC. The network from Chapter 2 is implemented succesfully. The most important decisions that had to be made are substantiated in Chapter 4 and 5.

The main results of these study are the protocols for $t = 1$ and the comparison for $t = 2$ proposed in Chapter 6. The results for $t = 1$ are summarized in the following table. Note that the calculation times are based on our experiments and that they might be different on other computers.

	trunc (removing f bits)	LSB	comparison$_{t=1}$(bit length ℓ)	PRandBit
old complexity	2 rnd, $f+1$ inv	2 rnd, 2 inv	$1+\lceil\log_2(\ell+2)\rceil$ rnd, $2\ell+2$ inv	1 inv, exponentiation
new complexity	1 rnd, 1 inv	2 rnd, 2 inv	3 rnd, $2\ell+3$ inv	1 inv
calculation time	75% decrease	30% decrease	37.5% decrease	?

Table 7.1: the new protocols for $t = 1$ suggested in Chapter 6 compared to the old protocols

Furthermore the round complexity of the comparison for $t = 2$ has been reduced from $1+\lceil\log_2 \ell+2\rceil$ to 4 rounds.

7.1 Future Research

- Investigate whether there are techniques for additive sharings that are more efficient than the corresponding VIFF protocols.
- Implement the new comparison for $t = 1$ with $n > 3$ and $t = 2$.
- Implement the new method of generating pseudo random bits for $t = 1$
- Implement the LSB protocol for $(t,n) = (1,3)$ with lower round compexity as suggested in Subsection 6.4.1

Bibliography

[1] Ramesh C Agarwal and C Sidney Burrus. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560, 1975.

[2] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 192–206, 2008.

[3] M. Augustine Cauchy. Méthode générale pour la résolution des systeèmes d'équations simultanées. *Comptes Rendus Hbd. Séances Acad. Sci.*, 25:536–538, 1847.

[4] Li Chen, Song Wang, Wei Fan, Jun Sun, and Satoshi Naoi. Beyond human recognition: A cnn-based framework for handwritten character recognition. In *3rd IAPR Asian Conference on Pattern Recognition, ACPR 2015, Kuala Lumpur, Malaysia, November 3-6, 2015*, pages 695–699, 2015.

[5] Yunpeng Chen, Jianan Li, Huaxin Xiao, Xiaojie Jin, Shuicheng Yan, and Jiashi Feng. Dual path networks. *CoRR*, abs/1707.01629, 2017.

[6] Sebastiaan de Hoogh. *Design of Large Scale Applications of Secure Multiparty Computation: Secure Linear Programming*. PhD thesis, University of Technology Eindhoven, 2012.

[7] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[8] Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In *Proceedings of the 10th International Conference on Practice and Theory in Public-key Cryptography*, PKC'07, pages 330–342, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[10] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017.

[11] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[12] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *Int. J. Inf. Sec.*, 14(6):531–548, 2015.

[13] 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.

[14] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 619–631, 2017.

[15] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[16] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[17] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[18] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[19] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.