FACULTY OF ENGINEERING TECHNOLOGY

DE NAYER (SINT-KATELIJNE-WAVER) CAMPUS

Secure face matching

Felix LERNER

Promotor: Prof. dr. ir. Toon Goedemé
Co-promotor: Dr. Pradip Mainali (Onespan)

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Industriele Wetenschappen Electronica-ICT

Academiejaar 2019 - 2020

Acknowledgements

This thesis could not have been written without the much appreciated help of other people.

I wish to show my gratitude to Toon Goedemé from Campus De Nayer, KU Leuven, for steering me in the right direction, assisting me during the study and for giving me useful feedback on my thesis. His assistance was a milestone in the completion of this thesis.

I would also like to pay my special regards to Pradip Mainali for helping me getting set up and giving me guidance. He got me intrigued with cryptography for which I am forever grateful. I admire his endless patience and I felt like the door was always open for when I had questions or when problems arose.

I would also like to thank all the other experts at OneSpan whose assistance was greatly appreciated. In particular; Tom De Wasch, Fabien Petitcolas and Marc Joye and ofcourse the IT guys for solving the many technical difficulties that I encountered.

Finally I would like to express my gratitude to my parents, my sister and my friends. They encouraged me during the years to pursue my technical studies. They have always been there for me and I know that I can count on them.

Thank you

Felix Lerner

Nederlandstalige Samenvatting

De (korte) samenvatting, toegankelijk voor een breed publiek, wordt in het Nederlands geschreven en bevat **maximum 3500 tekens**. Deze samenvatting moet ook verplicht opgeladen worden in KU Loket.

Abstract

We are affected with machine learning in many aspects of our daily lives, applications ranges from facial recognition to enhanced healthcare to self-driving cars. As companies outsource image classification tasks to cloud computing service providers, we see a rise in privacy concerns for both the users wishing to keep their data confidential, as for the company wishing to keep their classifier obfuscated.

We make use of existent cryptographic methods and state of the art computer vision algorithms to encrypt data in a way that an image classification task, like face matching, can be computed, while making sure that the privacy of the user is preserved.

Keywords: Computer vision, Cryptography, machine learning, secure multiparty computation, deep learning, object detetcion, privacy preserving, MLaaS, encryption

Contents

List of Figures

List of Tables

List of abbreviations

CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
MLaaS	Machine Learning as a Service
MPC	Secure Multiparty Computation
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
STDOUT	Standard Output
TLS	Transport Layer Security

1 Introduction

Deep learning-based object detection on images is a hot topic for researchers and interest in machine learning is steadily growing among miscellaneous businesses. Facial recognition is one of the many applications machine learning has to offer. A face recognition algorithm tries to recognise faces of the same person. Faces are very unique parts of our body, thus face matching can be used as a means to do biometric authentication. In this case, a client sends a picture containing their face to an external service which grants the client access if the face is similar to the one stored in the database.

More and more users are concerned about their privacy and the security of their data stored and processed on servers. Not only are they afraid of malicious hackers stealing their sensitive data, they also fear the servers operator will use their data for purposes other than the user agreed to. Big corporations have been found guilty of collecting user data for unethical purposes (source: Cadwalladr and Graham-Harrison (2018)).

Secure multiparty computation (MPC) is a subfield of cryptography, making it possible for a party to run an algortihm on confidential data, that is supposed to stay unknown even to the party running the algorithm. There exist different methods to perform privacy-preserving computations, MPC is the one we will use.

Secure Multiparty Computation and Machine Learning aren't new concepts, in fact they exist for over 40 years. But with the rise of big data and processing power lately, there has been an increase in research into these fields.

Because of these concerns researchers are looking for technologies to enhance the privacy of the user during the processing of it's data on a server.

Onespan[1] (formerly VASCO Data Security International, Inc.) is a global company were most of our research was done. Onespan offers a series of security and authentication products and technologies and specializes in digital identity and anti-fraud solutions. The company continues to be active in research and innovation in different fields of technology, especially cryptography and data science.

In this thesis we study the applicability of MPC protocols on deep learning-based face matching algorithms and try to implement a privacy-preserving face matching algorithm.

1.1 Problem

The use of third party MLaaS (Machine Learning as a Service) providers or any cloud computing solution, as processing power for an image classification task, raises privacy concerns as sensitive images of users need to be sent to servers running an instance of the neural network.

It's important to note that the transport of the image from the client to the server is deemed to be secure, since the parties can make use of reliable HTTPS (Hypertext Transfer Protocol Secure) connections.

The user's images, however, are stored in plaintext [2] on the server, as well as the computed output of the image.

Furthermore the whole design of the neural network including all trained parameters needs to be stored on the servers of the third party, for the image classifier to function. Both of these remote processing solutions require a considerably amount of trust in the third party. Since the third party could potentially exploit the user data for commercial purposes or even steal the intellectual property of the image classifier. Of course most cloud computing service providers are not inherently malicious. But as long as the user's data is stored in cleartext on the server, there is a risk that the service provider could turn malicious. Or even worse, hackers could break in to the server and breach the confidential user data.

In this thesis we try to tackle the need to trust a third party MLaaS provider. We want it to compute an encrypted image on an obfuscated neural network to ouput a correct encrypted result. This encrypted result shall be sent to the client, which will then decrypt it. Figure 1.1 shows an oversimplified overview of this algorithm.

[1]Onespan's official website: `https://www.onespan.com`

[2]plaintext or cleartext are common cryptographic terms for unencrypted data.

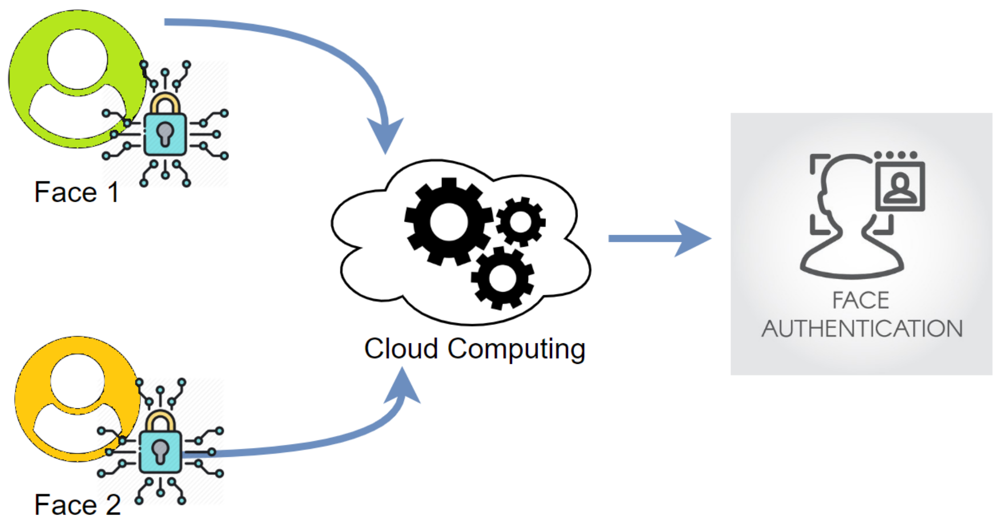


Figure 1.1: Overview of secure face matching. The cloud computing services compute over encrypted inputs.

Before we begin specifying our hypotheses, lets define some frequently occuring terms:

- Encryption: the process of converting data into a code, to ensure confidentiality and prevent unauthorized access.
- Privacy-preserving/secure algorithms: Algorithms that allow computation of private data, while preserving privacy.
- Face matching algorithm: Set of rules that a computer uses to compare two faces, to determine wheter there is a match.

1.2 Hypothesis

How can we securely compute the inference of a deep learning-based facial recognition neural network?
With the use of MPC protocols we can implement methods such that we can compute a whole face recognition convolutional neural network on an encrypted image of a face. This preserves the privacy of the user while allowing the computation to be outsourced to an untrusted third party.

How can we optimize the secure facial recognition task to run more efficiently?
We predict a drastic decrease in perfomance when running inference on the privacy-preserving neural network, because MPC is a protocol over a network of parties the computational time will not be the only factor to account for. We will try to find performance optimizations along the way

of implementing a proof of concept by looking at existing optimization concepts for MPC as well as optimization solutions for neural networks.

2
Literature study

In this chapter we will take a quick look at how convolutional neural networks (CNN) function, layer by layer. We will also learn how secure multiparty computation (MPC) in general is possible. Finally, we will discuss the related work on combining these two subjects so far.

The reason we don't just train our network on encrypted images as input data. Is because machine learning is generally based on discovering statistical patterns in data and the whole point of encryption is to make sure there is no statistical patterns and with truly random data there is none (think of image noise figure 2.1). In addition there is no heuristic you can use to tell if you are getting close to a correct classification.

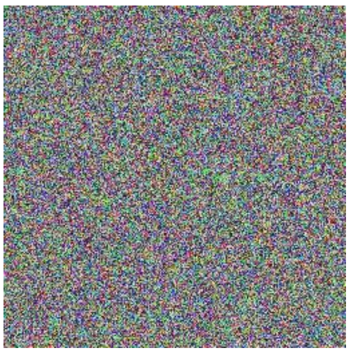


Figure 2.1: Random noise

2.1 Convolutional neural network

Convolution neural networks (CNN) is a special type of neural network used for images. The spatial properties of the pixels in the image are used during the evaluation of the input, meaning the neighbouring pixels of a central pixel impact the output to the next layer of that central pixel while pixels further away do not. CNNs are made of multiple layers. Typically, as you move further from the input layer to the output layer the dimensionality reduces, we can say the input gets mapped on a desired output manifold. Inputs that we classify as similar are supposed to be in the same region in the output manifold. Neural networks go through two phases: training and inference. A neural network needs to be trained in order to achieve good results. After the training the network will try to predict things based on the inputted data, in this phase the parameters of the network do not change.

2.1.1 Convolution layer

In this layer a discrete convolution of a kernel K shifting over an image I is performed, as shown in equation 2.1. The kernel has parameters also known as weights, so that certain features get extracted from the input.

$$(I * K)[m,n] = \sum_j \sum_k I[m-j, n-k]K[j,k] \tag{2.1}$$

The output of this layer is a convolved feature map. The parameters of the kernel are usually floating-point numbers and can be positive or negative.

2.1.2 Activation function

Since these convolutions are simple lineair operations and most image classification tasks require non-lineair classifiers, non-lineairity needs to be added to the neural network. This is achieved through adding a non-linear activation function after a convolution or fully connected layer. The most popular activation function is the rectified linear unit (ReLU) as seen in equation 2.2 and figure 2.2.

$$f(x) = max(0,x) \tag{2.2}$$

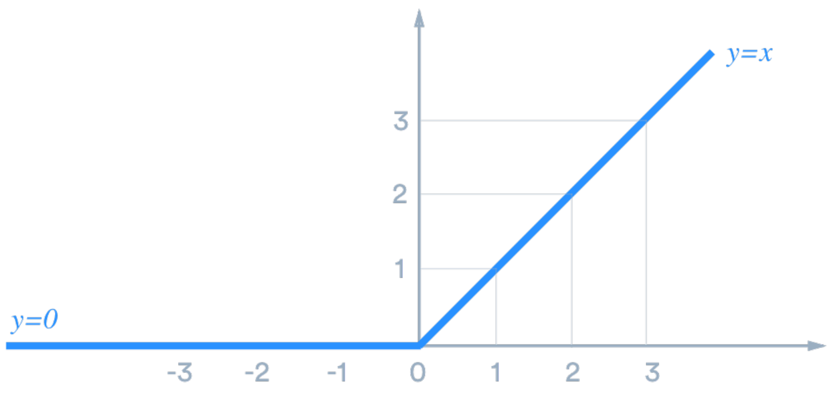


Figure 2.2: ReLu activation function

2.1.3 Pooling layer

The dimensionality reduction we talked about in the beginning of this chapter happens primarily in the pooling layer [1]. A kernel is shifted over the image. In the case of max-pooling the kernel selects the maximum value of the portion of the image it covers, to create the new dimensionality reduced image. A portion of the image value is theoretically lost, but because we only retain the maximum value in the window we sort of extract a feature from the input matrix. An example of this process can be seen in figure 2.3.

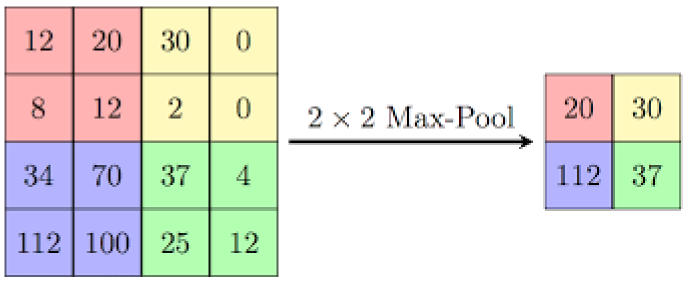


Figure 2.3: Example of 2×2 max-pooling

2.1.4 Fully connected layer

The fully connected layer is a multilayer perceptron that discriminates different object classes and identifies identical ones. All elements in vector h_{i-1}^{out} have their own bias B_i and weight W_i so that h_i^{in} can be calculated for each layer i according to equation 2.3.

$$h_i^{in} = h_{i-1}^{out} \cdot W_i + B_i \tag{2.3}$$

The fully connected layers usually come after the last convolution layer. The output of a convolution layer is a tensor, this means that the output needs to be flattened to a one-dimensional array. The

[1] Dimensionality reduction can also be combined with feature extraction in the convolution layer.

last layer of the fully connected layers is called the output layer. This output layer determines the classification.

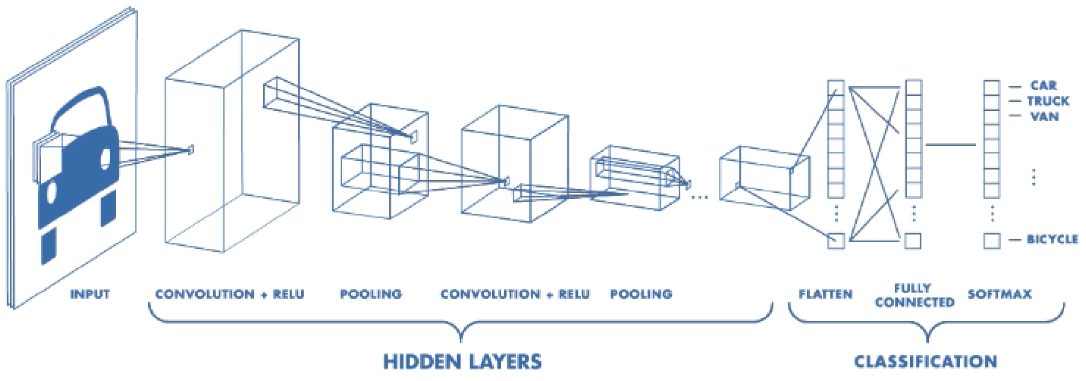


Figure 2.4: Overview of CNN

When all these layers are connected to each other, as you can see in figure 2.4, we speak of a model with a CNN architecture. And if there are enough layers between the input layer and the output layer, we say the model is a Deep Neural Network (DNN).

2.1.5 Face matching

Face matching is an algorithm that tries to match two faces of the same person. Face matching can be performed with convolutional neural networks an image gets fed to the network and produces an output vector depending on what face was in the image. Koch et al. (2015) showed that a siamese neural network makes it possible to not only recognise new data (unseen during the training) but to also recognise entirely new classes. In the case of face matching each person's face is a class. With siamese neural networks it is thus possible to recognise faces of persons which the network didn't see during training.

A siamese neural network consists of two CNN's. These CNN's are identical. A siamese neural network has to be fed two images, it then produces two output vectors. When these two images have faces that are similar the euclidean distance between the two output vectors will be small. When the two faces on the images are dissimilar, the euclidean distance between the two vectors is large. Thus to determine if two images are of the same person, we calculate the euclidean distance (equation 2.4) between the two output vectors, if the distance lies under a certain threshold we accept that the two faces belong to the same person. If one output vector of the two images is stored in a database, an application can use this algorithm to allow for biometric authentication. This output vector is also referred to as the embedding of the face or a point on a manifold. A manifold is a topological space that locally resembles the Euclidean space near each point, this means that each point of that n-th dimensional manifold has a neighborhood that is homeomorphic to the Euclidean space of dimension n. This is great news, we can compare two embeddings on the manifold by calculating the Euclidean distance between the two points, the only condition is that the two points are in eachothers locality or neighborhood. A general overview of a siamese neural network architecture can be found in figure 2.5.

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{2.4}$$

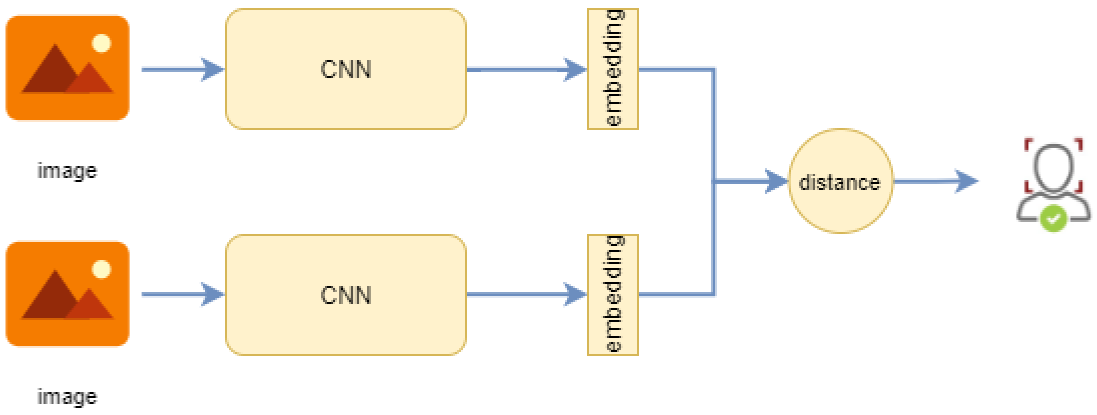


Figure 2.5: Overview of siamese neural network

2.2 Secure multiparty computation

Secure multiparty coputation is a protocol that is used between n number of parties P. Each of these parties has private data also called a secret S. With MPC it is possible for these parties to compute a public[2] function f on the secrets. Such that a party $P_i \in P$ only knows his secret $s_i \in S$ and the public securely computed output $f(s_0, s_2, s_{n-1})$ after the protocol has succesfully finished. A classic application is Yao's Millionaires' problem Yao (1982) in wich two millionaires wish to know who is richer, there is catch however. Instead of making their balances publicly known. They wish to keep their balances a secret. In this case the number of parties n is 2 the secrets s_0 and s_1 are their balances. The public function $f(s_0, s_1) = 1$ if $s_0 < s_1$ and 0 otherwise.

We categorize 2 types of parties based on their willingness to deviate from the correct predefined protocol.

- **Honest parties:** Parties do not wish to know other parties secrets and will never reveal the secret.
- **Honest but curious parties (passive):** Parties wish to know other parties secrets but will not deviate from the protocol at any time. Also called semi-honest parties.

[2]Public or global means known to all parties, while private or local means known only by the corresponding party.

- **Malicious parties (active):** Parties wish to know other parties secrets and wish to change output of computation to favourable result. Parties will deviate from the protocol to cheat and change the outcome at any time.

In practice the whole set of parties will exist of subsets of these different types of parties. Ideally every party is honest, but this is rather a naive way of thinking.

If the two millionaires are honest but curious parties, they will not deviate from the protocol and they will computer the correct output as a result they will know who is the richer millionair but they won't know how much money the other one has. In the other case one of the two millionaires is corrupt and acts maliciously, the honest millionair will follow protocol while the dishonest millionair will deviate from the protocol to change the result in his favour. In the event that the dishonest millionair is poorer he will change the outcome thus appearing richer. From now on, we assume the set of parties are a mix of honest and semi-honest parties, unless specified otherwise. We also assume the communication between the different parties to be secure.

The efficiency of an MPC protocol is defined by three metrics:

- **round complexity:** The amount of rounds needed in the protocol. In one round each party can read all messages sent to the party in the previous round as well as perform an arbitrary amount of local computation and finally send messages to all other parties.
- **communication complexity:** The amount of communication between all parties (measured in bits).
- **computational complexity:** The number of primitive operations performed by all parties.

In general, the computational complexity is very low while the communication complexity dominates the protocol's total complexity. Thus an estimate of the overall complexity can be measured by combining the round and communication complexity. This means the efficiency of MPC protocols is heavily based on the network's latency rather than the network's throughput.

2.2.1 Secret sharing

In order to do secure computing, the parties need to split their secret into secret shares. A secret sharing method can be used by the secret holder to split a secret into a number of shares. Combining these shares will reveal the secret, while individual shares alone will leak nothing about the secret. In a (t,n) threshold secret sharing sheme parties must combine at least t shares of the total n shares, to obtain the secret. We can now set a threshold t high enough, denying the secret to small curious parties and allowing to reveal the secret when a majority $(\geq t)$ consensus is reached. Shamir's secret sharing scheme Shamir (1979) is based on polynomial interpolation and the essential idea is that it takes at least t points in order to define a polynomial $p(x)$ of degree $t-1$. Given a set of t points in a 2-dimensional carthesian system $(x_1,y_1),(x_2,y_2),...,(x_t,y_t)$, there exists only one polynomial of degree $t-1$. This can be proven and the mathematical construction

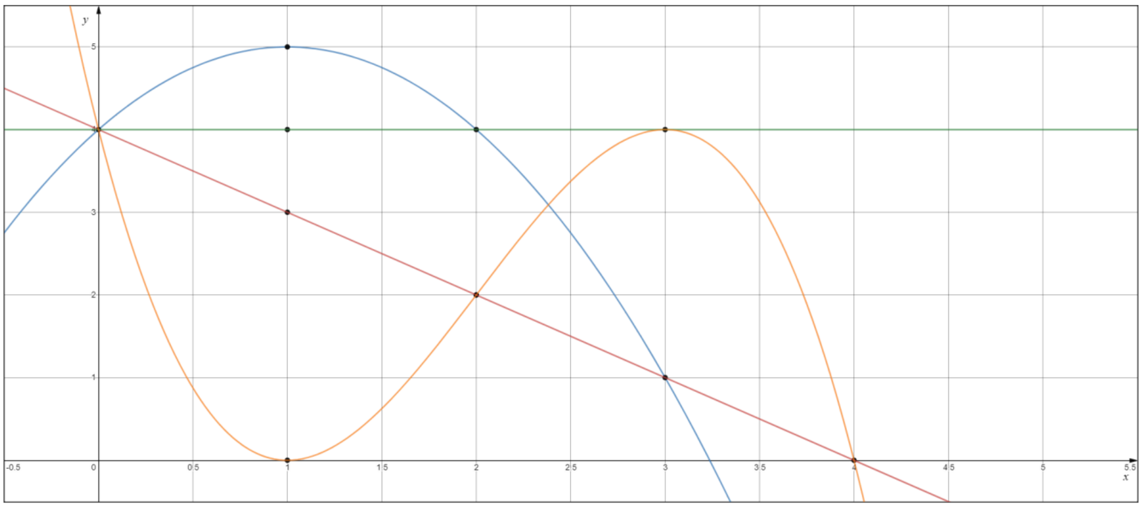


Figure 2.6: Shares for different t with secret $s = 4$

of a polynomial $p(x)$ of degree $t-1$ based on a set of t points can be calculated using Lagrange's interpolation formula 2.5.

$$p(x) = \sum_{i=1}^{t} y_i \delta_i(x) \quad \text{with} \quad \delta_i(x) = \prod_{1 \le j < t; i \ne j} \frac{x - x_i}{x_j - x_i} \tag{2.5}$$

With this in mind, a secret dealer can now share his secret s to n parties by choosing a random $t-1$ degree polynomial $p(x) = a_0 + a_1 x + ... + a_{t-1} x^{t-1}$ in which a_0 is the secret or the number representation of the secret if the secret is not a number. The dealer now calculates n points on the polynomial starting from $x = 1$, because the secret is located at $x = 0$. Each party $P_i \in P_1, P_2, ..., P_n$ is given a different single point (x_i, y_i), at this stage the secret is shared. The convention for the notation of a shared secret is $[s]$. We can say a party P_i is holding a share in the form of a point (x_i, y_i) or shortened $[s]_i$.

To recombine the secret, the parties simply broadcast or send their shares to a central entity, if more than t shares are known, it suffices to calculate the Lagrange polynomial $p(x)$ and $s = p(0)$. In the case of not having enough shares, the Lagrange polynomial containing the secret becomes impossible to calculate since every polynomial is equally likely, thus revealing absolutely nothing about the secret.

Note that all arithmetic in this section can be done over some finite field $\mathbb{F}_q$ to speed up the algorithms.

A Generalisation of this (t,n)-Shamir secret sharing scheme from thesis de Hoogh (2012) as follows (Protocol 2.1 and Protocol 2.2):

1. **Share Generation:** To share $s \in \mathbb{F}_q$, the dealer generates random $a_1, ..., a_t \in \mathbb{F}_q$ and puts $p(x) = s + a_1 x + ... + a_t x_t$. Then the dealer computes $[s]_i = p(i)$.

2. **Share Distribution:** For each $i \in 1, ..., n$, the dealer sends $[s]_i$ to party P_i.

3. **Secret Reconstruction:** Let $D \subset 1,...,n$ be a set of size $t+1$. Each party P_i for $i \in D$ sends his share $[s]_i$ to all parties. Then, each party reconstructs the secret via Lagrange interpolation.

In figure 2.6 a visualisation of this scheme shows us that in order to get the secret value, $s = 4$ in this case, we need to know at least $t+1$ points for a given t. Two points are needed to recombine a first order polynomial (red line), three points for a second order polynomial and so on.

It's important to make sure the parties are distributed and to try minimizing the incentive to collude. Distribution is needed to lower the risk of having a malicious party taking over control of the network, this can be done by splitting the parties up to many small stakeholders instead of a couple centralized stakeholders. These stakeholders could be rival companies, local governments or even individual citizens. The only way to totally avoid collusion, is to let the secret holder partake to the MPC. However, this is not wished, as we want the computation to be outsourced.

2.2.2 Operations

A neural network can be seen as an enormous function with millions of coefficients. Lucky for us the function can be broken up into 3 different operations: addition and multiplication in fully connected and convolution layers and a relational operator for max pooling and ReLU activation function. We will now show how to securely compute each of these basic operations.

2.2.2.1 Arithmetic operators

Addition, subtraction, division and multiplication all fall under arithmetic operations. In this section we will take a look at how these operations can be performed in MPC.

linear protocols: Since Shamir's secrete sharing scheme is a linear sharing scheme, each party P_i can locally compute any linear combination of a public or secret value with their secret share $[s]_i$. This gives us following operations:

- **Addition of secret and public value ($[c] \leftarrow [a] + \beta$):** Each party P_i locally adds the public value $\beta \in \mathbb{F}_q$ to it's share $[a]_i$, resulting in the new share $[c]_i = [a]_i + \beta$. Since all parties add the same β, this value is a constant.
- **Multiplication of secret and public value ($[c] \leftarrow [a] \cdot \beta$):** Each party P_i locally multiplies the public (constant) value $\beta \in \mathbb{F}_q$ with it's share $[a]_i$, resulting in the new share $[c]_i = [a]_i \cdot \beta$.
- **Addition of multiple secrets ($[c] \leftarrow [a] + [b]$):** Each party P_i locally adds it's secret shares $[a]_i$ and $[b]_i$, resulting in the new share $[c]_i = [a]_i + [b]_i$.

The last operation is visually demonstrated in figure 2.7. In this example secret $a = 1$ (green) and secret $b = 3$ (red). After each party P_i locally computes the addition of $[a]_i$ and $[b]_i$ and stores it as a new share $[c]_i$. After broadcasting the new, computed share $[c]$, they can recombine the shares

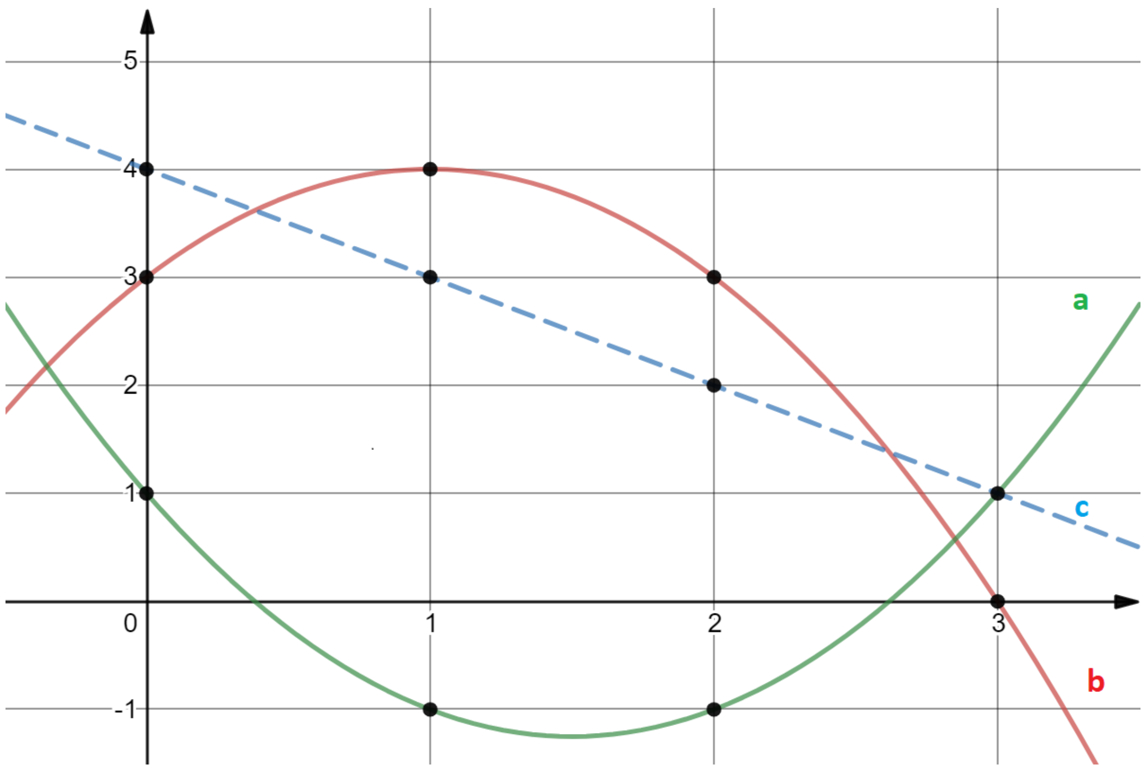


Figure 2.7: Adding two secrets for $n = 3$ parties

via Lagrange interpolation to get the polynomial of the combined shares $[c]$ and more importantly the output of the addition of the two secrets a and b. This happens all with zero knowledge about a or b. All of the computations are done locally and no communication other than the inital share distribution and the share reconstruction is needed.

The inverse operations (subtraction and division) are possible too, and the protocol is the same.

Multiplication protocol: Multiplication of two secret values is more challenging since it isn't a linear operation. Their are multiple methods to perform secret multiplication, the method described by Ben-Or et al. (1988) also called the BGW protocol (initials of the authors) has to solve two problems along the way of computing $c = a \cdot b$.

Assume a and b are respectively encoded by $f(x)$ and $g(x)$ and $n \geq 2t+1$. The free coefficient of $h(x) = f(x) \cdot g(x)$ is simply $h(0) = f(0) \cdot g(0)$, this means a simple multiplication of the polynomials would be sufficient to compute the multiplication of the secrets. There is however a major problem, multiplication of two polynomials of degree t yields a polynomial of degree $2t$.

While this poses no problem with interpolating $h(x)$ from it's shares $[c]$ since $n \geq 2t+1$, further multiplications will continue to raise the degree to a level where $t > n$ making it impossible to interpolate the resulting polynomial $h(x)$. The second problem is harder to spot. The polynomial $h(x)$ as a result of the multiplication of polynomials $f(x)$ and $g(x)$ is reducible (since it's a multiplication). In other words $f(x)$ and $g(x)$ are uniformly random polynomials of degree t, but $h(x)$ as a multiplication of two random polynomials is not irreducible, therefore $h(x)$ is not uniformly random. A uniformly random polynomial is a polynomial with coefficients that are randomly sampled according to the uniform distribution, i.e. the coefficients are randomly sampled from a set where drawing each element is equally probable. To make sure the resulting polynomial stays of degree t and is uniformly random, the parties run a protocol to generate a random polynomial of degree $2t$.

The result of a multiplication of a uniformly random polynomials of degree t with a uniformly random polynomial of degree less than or equal to t is a polynomial of degree $2t$. Assuming the base field is Z_q, then there are q^{2t+1} such polynomials, but we would only get q^{t+1} polynomials. So the distribution is not uniform.

This protocol works as follows. Each party P_i randomly selects a private polynomial $q_i(x)$ with secret $q_i(0) = 0$ of degree $2t$ and distributes it's shares among the parties. Each party P_i now has n random shares $[q]_i^k$ (with $k : 1 \rightarrow n$). After each party adds all it's random shares they hold a secret, truly random polynomial $q(x)$ with a zero as free coefficient $q(0) = 0$. Each party P_i now computes the multiplication of the two secret values $[a]$ and $[b]$ and instead of using $[c]$ encoded as $h(x)$ we can add the random polynomial to the result, thus randomizing the coefficients and making the polynomial uniformly random. This will not mess up the result as now $a \cdot b$ is $(f(0) \cdot g(0)) + q(0)$ and $q(0)$ equals zero. This step is also called the randomization step.

The parties now run a protocol to reduce the degree of $h(x)$ to t. This protocol can be computed locally (no communication is required) by multiplying the shares of the polynomial $h(x)$ to a specific, matrix of constants. This will truncate the result $h(x)$ of degree $2t$ to a polynomial of degree t. Proof for this degree reduction step can be found in the study by Asharov and Lindell (2017).

After these two steps, the parties hold $c = a \cdot b$ encoded by $h(x)$ of degree t with coefficients uniformly distributed. They can now recombine their shares to find the polynomial $h(x)$ and the product of a and b, $h(0)$. The whole multiplication protocol requires only one additional round of communication, this happens during the distrubition of the shares of the random polynomials $q(x)_i$ in the randomization step. Note that since we conditioned the protocol to work in cases where $n \geq 2t + 1$, a majority of honest parties is needed. Opposed to the linear protocols explained earlier, where only one party needed to be honest.

A large proportion of the neural network is ready to be transformed with these secure arithmetic operators, namely the convolution layer and the fully connected layer.

2.2.2.2 Relational operators

In this section we will focus on comparison between secrets. There are two important protocols in secure comparison, equality testing and greater-than testing.

Equality testing: Suppose we have two shared secrets $[a]$ and $[b]$ and the parties want to know if $a = b$, without knowing a or b. The easy way, would be to just securely compute and reveal $c = a - b$. This would however reveal secret b if a was to be revealed, since $b = a - c$. To make sure the the output of the subtraction is irreducible and uniformly random Franklin and Haber (1996) came up with an idea to let the parties generate a random shared non-zero secret $[r]$. Then compute and reveal $c = (a-b)\cdot r$. Since $r \neq 0$, if $c = 0$, a must be equal to b. If $c \neq 0$, a and b are not equal. If we want to compare a secret with a public value β, we just take a as the secret and use a β instead of $[b]$ in the protocol. In this case the naive method $c = (a-\beta)$ would just give away the secret value even if $a \neq \beta$, so it's absolutely required to use a random multiplier r to hide the reversible operation.

The generation of a random shared non-zero secret $[r]$ appears to be very similar to the generation of the random share in the randomization step of the multiplication protocol but there is one difference, in this case the secret value must be different than zero (invertible), while in the multiplication protocol the random share needed to be equal to zero. The idea is to generate two shared random secrets $[x]$ and $[y]$. Then the parties compute the product of the two secrets $z = x \cdot y$ and reveal z. If $z \neq 0$, both random secrets x and y are non-zero, thus applicable in the equality testing protocol. If $z = 0$ repeat the random share generation protocol with different random shared secrets and retry the multiplication with these new shares.

Greater-than testing: Often we want to know more about two secrets than just equal or different. We want a protocol comparing two secrets a and b that returns a boolean for $a > b$. There exists multiple different protocols for greater-than comparisons. The one we use in our implementation is published in Erkin et al. (2009). This protocol compares the two secrets on bit-level.

2.2.3 Number representation

When a CNN gets computed on cleartext data almost all parameters of the CNN are floating-points, it is thus important that these numbers can be transformed to representations suitable for MPC.

Floating-points can be implemented in MPC. But they usually come with a high complexity cost for addition as well as multiplication and comparison tests. The advantage of floats over other number representations like scaled integers and fixed-point numbers is that the the maximal rounding error scales with the magnitude of the number. Unlike floats fixed-points have a rounding error of a fixed size. Campmans (2018) studied the use of fixed-point numbers as an alternative for floating-point numbers in MPC protocols. He states that with fixed-point numbers no comparison needs to be done during multiplication of two secrets. While the costly comparison is needed when multiplicating two secret floating-point numbers.

A fixed point data type is essentialy an integer that is scaled by a chosen factor. Let's say our numbers in the MPC protocol never go out of the bounds $[-500, 500]$ and we want a precision of 2 fractional digits. Then we can use the integers bounded by $[-50000, 50000]$ to perform all arithmetic. The arithmetic operations will be the same as when we use integers. But we will actually be computing arithmetic operations on fixed-point numbers. This allows for an efficient approach to floating-point numbers while minimizing rounding errors. Of course we need to make sure that the operations don't overflow and that precision loss is small.

2.3 Overview

We now have seen the miscellaneous protocols MPC has to offer. From now on we will treat the seen MPC protocols like black boxes that accept inputs in the form of secret shares and compute outputs in the form of secret shares. As long as the parties do not collude to find the secret, the protocol is secure.
Overview of protocols:

- Secure addition of two secrets or a secret and a public value (F_{add}).
- Secure multiplication of secret and public value ($F_{mul}^{constant}$).
- Secure multiplication of two secrets (F_{mul}).
- Equality testing of two secrets or a secret and public value (F_{eq}).
- Greater-than testing of two secrets (F_{gt}).

This set of protocols can be used to transform any basic CNN to a secure one.

In figure 2.8 a high-level workflow of secure face recognition is shown. The MPC protocol works for three parties [3], who each have their own computing instance connected to the MPC network and the client. The workflow commences by acquiring an image of the clients face, this can be done by taking a photograph with the front-facing camera of the clients smartphone. The client then performs secret sharing on the image [4] and sends the shared secret to the participating parties. The parties receive their shares and jointly compute the output of the face recognition model on the given shared image of the face. The face recognition model can be public or secret. In the case of a secret model, the secret shares of the weights and biases need to be sent to the participating parties as well. Note that this only has to be done once, during initialisation. After the computation is finished, the parties send their resulting secret shares to the client. The client interpolates the shares to find the secret values determining if her face matches or not.

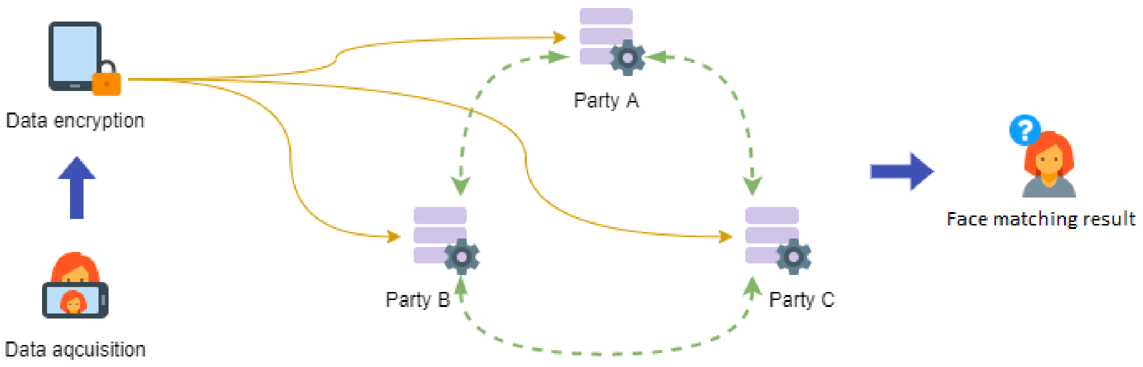


Figure 2.8: Workflow of secure face matching

[3]This is the smallest amount of parties needed in order to do secure multiplication (F_{mul}).
[4]Secret sharing is performed for each pixel of the image.

This protocol ensures that the client is the only one having knowledge about the cleartext of the image and the output of the face recognition algorithm. This protocol also offers an obfuscation of the model's parameters, the parties and the client have no knowledge about the parameters if the owner of the model uses secret sharing to send the parameters to the parties.

2.4 Related work

Erkin et al. (2009) presented for the first time a privacy-preserving face recognition alogithm using MPC, this was before the rise of machine learning. Thus they were restricted to using eigenvalues of the faces described in Turk and Pentland (1991). They show that their privacy-preserving face recognition algorithm is as reliable as the normal face recognition algorithm. Unfortunately, the computational cost for producing the eigenvalues of an image is way less than producing a result for a CNN.

Mainali and Shepherd (2019) recently published the first framework for privacy-ehancing fall detection from a body-worn inertial measurement unit using traditional machine learning and MPC. The data they work with is time-series inertial measurements this has a significantly lower dimensionality than an image. Traditional machine learning classifiers require less computations than deep neural networks. The authors hint to secure video-based fall detection as a possibility future research and to which extent secure video-based fall detection can be performed real-time with high-dimensionality.

Gilad-Bachrach et al. (2016) discuss implementing CryptoNets, a fully homomorphic encryption (FHE) based approach for privacy-preserving optical character recognition task on the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits. FHE is another approach to making machine learning privacy-preserving. But since we focus on MPC in this thesis, we refer the interested reader to the paper written by Gentry et al. (2009).

Barni et al. (2006) presented an algorithm that does privacy-preserving computation on a neural network using MPC. The data owner encrypts the input, via secret sharing, and sends the secret shares to the participating MPC parties. The parties compute the inner product of the weights of the layer and the data. The product gets sent back to the data owner. The data owner then adds non-linearity via an activation function on the outputs, this is done in cleartext since the data owner can decrypt the secret shares and is allowed to see is own data. The result then gets encrypted again and is sent to the MPC parties to compute the inner product in the second layer. This protocol is done for all the parties. In this algorithm the data owner communicates multiple times with the MPC parties. Some computation of the neural network is also done by the data owner. A strong cooperation between the parties and the data owner is thus needed.

Campmans (2018) worked on optimization of existing MPC protocols for convolutional neural network operations. He also suggested using fixed-point numbers instead of floating-point, losing some accuracy for more efficiency. He also explored the use of discrete fourier transforms as an approach to costly convolutions in MPC. The convolutions are costly because they require lots of multiplications.

Makri et al. (2019) proposed EPIC an efficient private immage classification system based on support vector machine (SVM) learning. They focus on efficiency by minimizing the load on the privacy-preserving part. Their experiments conclude that their is a tradeoff between efficiency and accuracy of the privacy-preserving image classification systems. They consider the deployment of a CNN with current MPC protocols in an active adversary MPC environment to be computationaly prohibitive. It should be noted that in our implementation, we assume the MPC environment to be made of passive adversaries.

Taigman et al. (2014) presented DeepFace, a face recognition algorithm trained on millions of images of Facebook users. It achieves a stunning 97.35% on the Labeled Faces in the Wild (LFW) dataset, which is the classic benchmark dataset for face recognition, closing the gap to human-level perfomance. The authors made use of a siamese neural network. But this alogrithm is already outdated. Deep learning-based networks use more layers than a regular neural network, Wang and Deng (2018) give an overview of the recently published alogrithms, most of them use a deep learning-based architecture. Some of the alogrithms achieve an acurracy of around 99.8% on the LFW dataset. This surpasses the ability of face recognition by humans. These neural networks often have millions if not billions of parameters. This means that the computational complexity is very high and it is unlikely that we will find a way to efficiently make this type of neural networks secure. Instead we will focus on slightly lighter neural networks with less layers and parameters.

2.5 Conclusion

It is theoretically possible to implement a privacy-preserving CNN that works just like a normal CNN on cleartext data. But in practice we expect some resistance based on the related work, the main problem is the complexity that arises when performing MPC protocols on high dimensional data like images. There is a tradeoff between precision and complexity. Our first steps shall be to implement a naive design of the MPC protocol for CNN's. Then we will try to improve the efficiency, by adding existing optimization solutions, thus minimizing the time it takes to perform one face matching task. We will compare our results with other privacy-preserving techniques for CNN's.

3 Implementation

In this chapter we will explain how we implemented MPC for face matching algorithms. We will do this by giving a high-level overview of the system and then diving deeper in more interesting parts. We also show some techniques to lose some complexity without losing too much accuracy. With the information in this chapter and the code in the appendix, you should be able to reproduce our experiments. Feel free to checkout our code by cloning our GitHub repository[1].

3.1 Specifications

There are two major subprojects. The first subproject (chapter 3.1.1) is making sure we can generate the appropriate parameters for the face matching network. It is important that the model is accurate enough. The second subproject (chapter 3.1.2) is about transforming the classic machine learning functions to secure ones. To add this security or privacy-preserving factor we use a MPC framework.

3.1.1 Deep Learning

A machine learning project usually includes on of the popular frameworks available to the public. Since we were already familiar with Pytorch we used this library as a python package.

Pytorch [2] provides us with a deep learning research platform that provides maximum flexibility and speed. It's fairly easy to use but that doesn't mean we can't design more complex models or fea-

[1]`https://github.com/Fluxmux/securefacematching`

[2]`https://pytorch.org`

tures. Pytorch uses tensors, tensors are multi-dimensional matrix containing elements of a single data type. Designing a neural network with Pytorch is as simple as defining a class with the layers in the correct order. An example of a neural network written using pytorch can be seen in the following code (listing 3.1)

Listing 3.1: Pytorch neural network example

```
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()

        self.cnn = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(1, 16, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(16),
            nn.MaxPool2d(kernel_size=2, stride=2),
            ...
        )
```

Making an accurate face matching neural network. Is a process that involves three major steps.

First of all the design or architecture of the network gets chosen. There exist a number of different topologies used in deep neural networks. But often choosing which one to take and how many layers to use, is the most difficult task. We will cover the architecture of the model in chapter 3.2.1. Adding more layers is the same as adding more parameters. And a model with more parameters is more complex.

The second step is called the traing of the neural network. Training is done using a part of the dataset that is specific for training and shouldn't be used for anything else.

A typical workflow of the training step looks something like this: Two labeled faces are sent seperatly sent through the network. The euclidean distance (equation 3.1) for inputs $\vec{X_1}$ and $\vec{X_2}$ and the parameterized function G_W calculates the distance between the outputs. This distance metric should be close to zero for faces belonging to the same person. But as large as possible for faces belonging to different person.

$$D_W(\vec{X_1},\vec{X_2}) = \|G_W(\vec{X_1}) - G_W(\vec{X_2})\| \tag{3.1}$$

Then we use the loss function Yann LeCun first introduced in his paper Hadsell et al. (2006); The general loss function L is the sum of contrastive loss functions for a training pair in the set of training pairs of size P.

Let Y be the binary label assigned to this pair of faces, $Y = 0$ if $\vec{X_1}$ and $\vec{X_2}$ are labeled as similar, and $Y = 1$ if $\vec{X}_1$ and $\vec{X}_2$ are labeled as dissimilar.

$$L(W,Y,\vec{X}_1,\vec{X}_2) = (1-Y)\frac{1}{2}(D_W)^2 + Y\frac{1}{2}(max^2(0,m-D_W)) \tag{3.2}$$

The contrastive loss function is definded in formula 3.2, where $m > 0$ is the margin. The margin m can be seen as a radius so that dissimilar pairs still contribute to the loss function if their distance is withing this radius. The goal of training is to minimize the loss. For similar pairs this means decreasing the distance D_W, for dissimilar pairs this means increasing the distance to be greater than the margin. We came to the conclusion that our loss reduces the most if we choose a margin of $m = 2$.

We also tried a different approach, instead of using this distance metric (which we would than have to statically threshold) we would like to have a probability value between 0 and 1. The output layer of the network consists out of two neurons (two classes), one neuron describes the similar faces, the other neuron describes the dissimilar faces. A probability of 0 means that the faces belong to different persons and a probability of 1 means that the faces belong to the same person. Binary Cross Entropy loss, or Logaritmic loss (equation 3.3) is one of the loss functions that can be used to achieve this.

$$L = -(y\log(p) + (1-y)\log(1-p)) \tag{3.3}$$

Let y be the binary label assigned to this pair of faces, p is the predicted probability. To visualize this loss function, have a look at figure 3.1

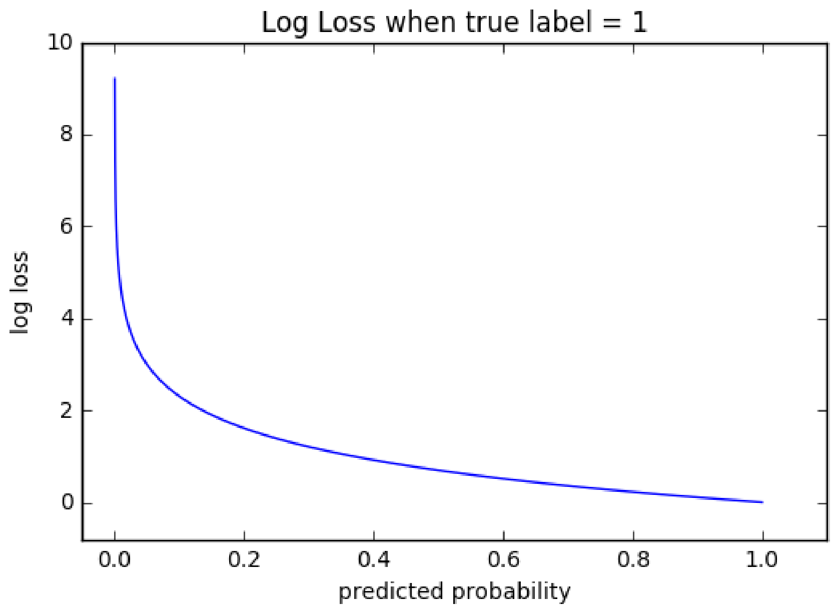


Figure 3.1: Range of loss values for $y = 1$

There exist a great number of other functions. Some might be better suited for the face matching problem. But since we need to keep things simple in order to implement MPC. We opt for one of the two loss functions described above.

Training a neural network takes some time, but the process can be sped up by using graphics processing units (GPU). We were lucky enough to have a dedicated GPU server at our disposal. While training the network is an easy task, we should look out for overfitting or underfitting.

Overfitting a model happens when there are too much parameters for a model or when the training was performed for too long. It will perform poorly on the validation dataset (part of dataset used to detect bad trainig behaviour) while performing excellent on the training dataset. There are two ways of overcoming overfitting. One way is to make the training dataset larger. Having more samples to train on, generalizes the learning model better. The other way is to design the model with fewer parameters, making it less complex. We use learning curves to track the training process of our face matching algortihm. An example of a more or less correct learning curve can be found in figure 3.2.

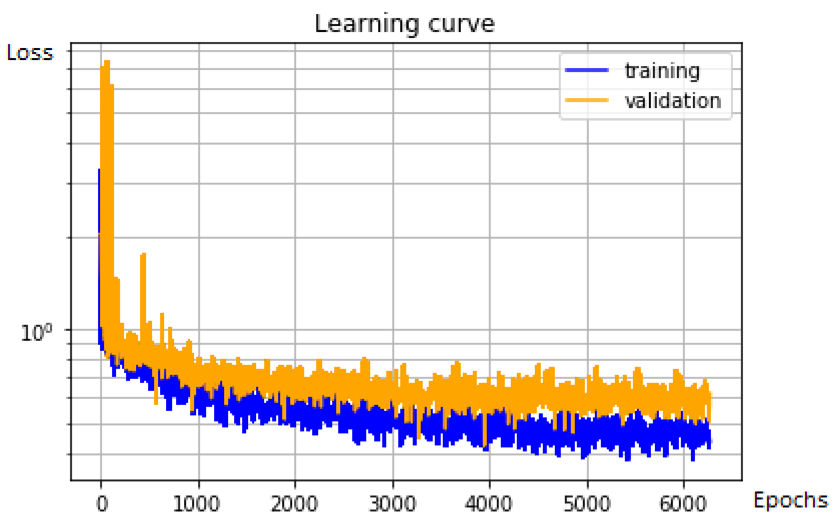


Figure 3.2: Learning curves are used to track the training of a model

Underfitting happens when a machine learning algorithm cannot capture the underlying structure of data. The model can't fit the data enough. Underfitting is more difficult to spot but easier to overcome. We can overcome this problem by making our model more complex.

As you can see by now, the architecture of a model is extremely important for it to function as wished.

The third step also called the hyperparameter tuning or hyperparameter optimization step, is what makes a good machine learning model even better. This process is not at all logic, experience and intuition can facilitate this. Hyperparameters are all the parameters whose values are set before the learning process begins. There are different methods for optimizing hyperparameters, since we wanted to learn the model and how it behaves relating to small changes in the values of the hyperparameters we went for Manual Random Search. Note that this took us some time because this involves multiple training steps. But since we had a GPU server at our disposal we could parallelize this task. In our case this step improved our accuracy by about 5%.

We used the Database of Faces[3] to train and test our face matching algortihm. The dataset contains a set of 10 images of frontal faces with different expressions per person and a total of 40 persons. The pictures are in pgm format which is extremely easy to interact with. They have a dimension of 1 x 92 x 112. An example of a set of images from the dataset can be seen in figure 3.3. We divided the dataset in to 3 parts: 75% training, 12.5% testing and another 12.5% for validation.

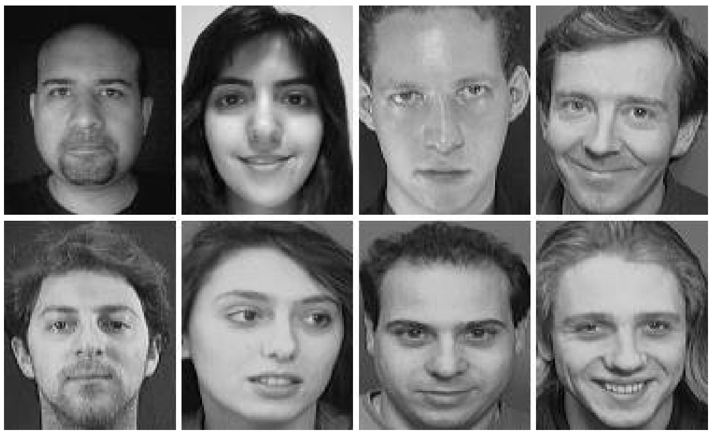


Figure 3.3: Example of faces in dataset

3.1.2 Secure Functions

Secure functions or the privacy preserving equivalent of an ordinary function can be used to compute an encrypted output as a function of encrypted inputs (figure 3.4). The protocol used for defining these secure functions is the MPC protocol.

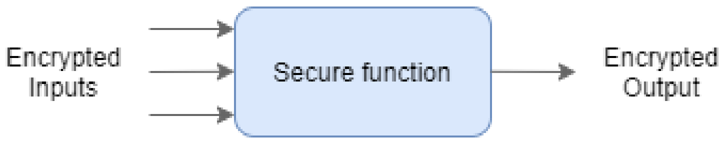


Figure 3.4: Secure function as a black box

Berry Schoenmakers is a cryptographer working on cryptographic protocols for electronic voting, electronic payments and secure multiparty computation. Since 2018 he has been working on a general MPC implementation for python, called MPyC. This pyhton package can be easily installed with pip: `pip install mpyc`. The mpyc package defines two new number representation types: SecFxp and SecInt. SecFxp are secure fixed-point numbers. SecInt are secure integers. For more information on how to use this package have a look at the source code at `https://github.com/lschoe/mpyc`.

[3]`http://cam-orl.co.uk/facedatabase.html`

There are number of basic functions predefined, like `add`, `sub`, `mul` and `div`, respectively $+$, $-$, $\times$ and $\div$. As well as `eq` (equal to), `ge` (greater than or equal), `min` and `max`. The total set of available functions can be found in `mpyc.runtime.py`. But for our task the functions noted above are all we need.

With this set of fundamental functions we can generate our own custom functions that serve our needs.

3.1.2.1 Custom Operations

We introduce three essential custom secure functions that can be found in `custom_operations.py`.

- `convolution`
- `maxpool`
- `relu`

`convolution` is a secure function that takes two arguments X and W as input as can be seen in listing 3.2. Let W be the kernel and X the image over which to do the convolution. X can be of any shape (m,n), W needs to be a square matrix of size s. We then define Y to be the output matrix of the convolution function. Because the convolution is padded, Y is of the same size as X.

Listing 3.2: Secure convolution function

```python
def convolution(X, W):
    m, n = dim(X)
    s = len(W)
    s2 = (s - 1) // 2
    Y = [None] * m
    for i in range(m):
        Y[i] = [None] * n
        for j in range(n):
            t = 0
            ix = i - s2
            for di in range(s):
                if 0 <= ix < m:
                    jx = j - s2
                    for dj in range(s):
                        if 0 <= jx < n:
                            t += X[ix][jx] * W[di][dj]
                        jx += 1
                ix += 1
            Y[i][j] = t
    return Y
```

Note that the number representation types that will be used in this secure convolution function, are not simple integers or floats. X, W and Y are matrices of SecInt's or SecFxp's.

`maxpool` is a secure function (listing 3.3). The function takes one argument as input, X an image of size (m,n). We define a specific type of max pooling, the pooling window has a size of 2 by 2 and the stride is equal to 2. The output Y is a matrix of size $(\lfloor \frac{m}{2} \rfloor, \lfloor \frac{n}{2} \rfloor)$.

Listing 3.3: Secure max pooling function

```
def maxpool(X):
    m, n = dim(X)
    Y = [None] * (m // 2)
    for i in range(0, m - 1, 2):
        Y[int(i/2)] = [None] * (n // 2)
        for j in range(0, n - 1, 2):
            Y[int(i/2)][int(j/2)] = mpc.max(X[i][j], X[i][j+1],
               X[i+1][j], X[i+1][j+1])
    return np.array(Y)
```

`relu` is a secure function (listing 3.4). The function takes one argument as input, X the image of size (m,n). This function doesn't change the shape of the input matrix, only the value of the elements of the matrix are changed, if they fulfill the condition.

Listing 3.4: Secure ReLU function

```
def relu(X):
    return np.vectorize(lambda a: (a >= 0) * a)(X)
```

Note that writing `a >= 0` or `mpc.ge(a, 0)` has the same effect. The same can be said for `(a >= 0) * a` and `mpc.mul(a >=0, a)`. The MPC operators are wrapped for the basic python operators on SecInt's and SecFxp's.

These three basic building blocks are essential for a convolutional neural network. But for the MPC protocol to function properly, we need to address some other problems. We will continue by giving a practical overview of the whole MPC protocol.

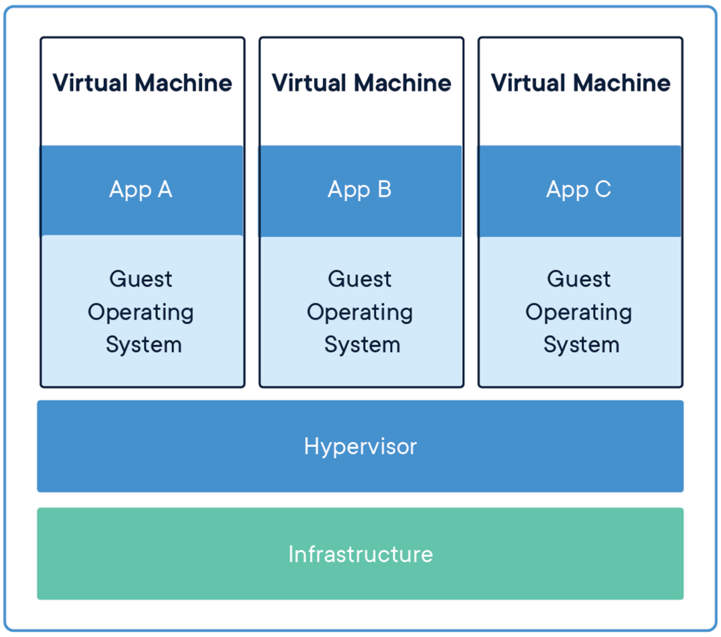


Figure 3.5: Three docker containers sharing the same infrastructure

Suppose we have one user (device) and three computing parties (server)[4]. We use the terms device and server to differentiate between two groups. On one hand we have the users who request a secure face matching task to be done, these users belong to the device group. On the other hand we have the parties doing the MPC, they belong to the server group.

The parties can be simulated by using three different Docker containers (figure 3.5), these containers share the same infrastructure (network). By setting the containers to different ports (5000, 5001, 5002 and 11365, 11366, 11367) on this network we can access, the containers from outside (device)[5]. The Docker file (listing 3.5) and environment file (listing 3.6) for setting up the three parties.

Listing 3.5: Docker files for servers

```
FROM tsutomu7/scientific-python
MAINTAINER Carlton Shepherd "carlton.shepherd@onespan.com"
COPY . /app

# Install requirements
WORKDIR /app
RUN pip install -r requirements.txt

# Install MPyC from source after giving
# default user (scientist) all ownership
```

[4]Note that for ease, we use a loopback interface (localhost)

[5]With outside we mean outside of the other containers, but the machine accessing the containers must still be on the loopback interface

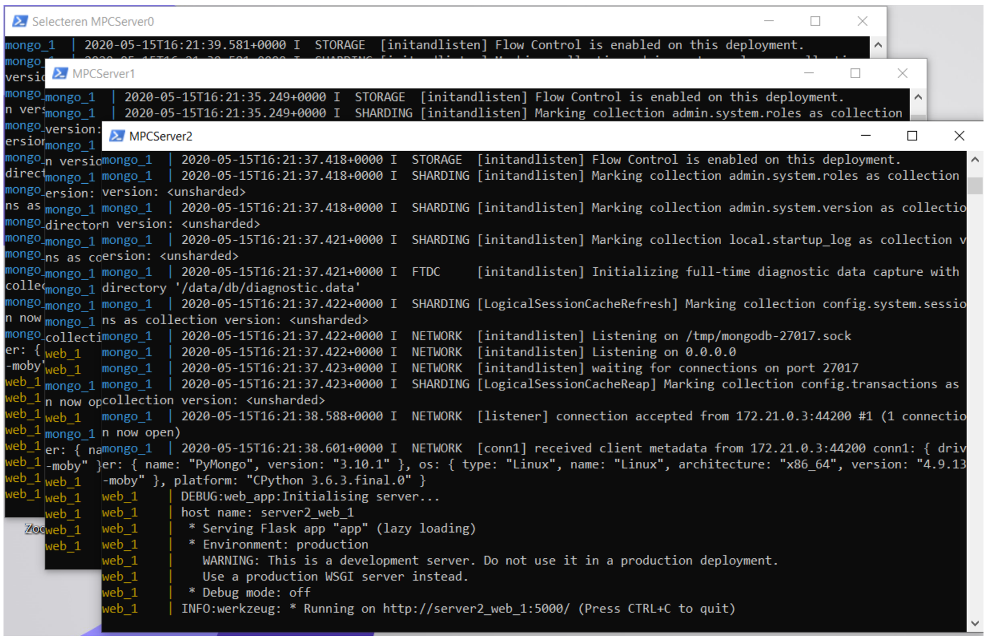
```
WORKDIR ./mpyc
USER root
RUN chown -R scientist .
USER scientist
RUN python setup.py install --user

#EXPOSE 11365

# Launch server
WORKDIR ../
CMD ["python -u", "./app.py"]
```

The `app.py` script runs a webserver using the python package flask and a database using the python package pymongo (MongoDB). The webserver will be used to communicate between the device and the server (to launch the MPC protocol, clear the database, ...) while the MongoDB will be used to store the secret shares for that particular server.

Listing 3.6: Environment file for servers

```
N_PARTIES=3
PARTY_0_HOST=localhost
PARTY_0_PORT=5000
PARTY_1_HOST=localhost
PARTY_1_PORT=5001
PARTY_2_HOST=localhost
PARTY_2_PORT=5002
```

When all three parties (server) are up and running as can be seen in figure 3.6, the user (device) can start encrypting their input image containing their face using Shamir's secret sharing scheme (listing 3.7). The user shares each pixel of the image in to three secret shares, one for each party. The user then sends their three sets of secret shares (encrypted image) to the parties, each party receives only one set of shares.



```
mongo_1  | 2020-05-15T16:21:39.581+0000 I  STORAGE  [initandlisten] Flow Control is enabled on this deployment.
```

```
mongo_1  | 2020-05-15T16:21:35.249+0000 I  STORAGE  [initandlisten] Flow Control is enabled on this deployment.
mongo_1  | 2020-05-15T16:21:35.249+0000 I  SHARDING [initandlisten] Marking collection admin.system.roles as collection
```

```
mongo_1  | 2020-05-15T16:21:37.418+0000 I  STORAGE  [initandlisten] Flow Control is enabled on this deployment.
mongo_1  | 2020-05-15T16:21:37.418+0000 I  SHARDING [initandlisten] Marking collection admin.system.roles as collection
version: <unsharded>
mongo_1  | 2020-05-15T16:21:37.418+0000 I  SHARDING [initandlisten] Marking collection admin.system.version as collectio
n version: <unsharded>
mongo_1  | 2020-05-15T16:21:37.421+0000 I  SHARDING [initandlisten] Marking collection local.startup_log as collection v
ersion: <unsharded>
mongo_1  | 2020-05-15T16:21:37.421+0000 I  FTDC     [initandlisten] Initializing full-time diagnostic data capture with
directory '/data/db/diagnostic.data'
mongo_1  | 2020-05-15T16:21:37.422+0000 I  SHARDING [LogicalSessionCacheRefresh] Marking collection config.system.sessio
ns as collection version: <unsharded>
mongo_1  | 2020-05-15T16:21:37.422+0000 I  NETWORK  [initandlisten] Listening on /tmp/mongodb-27017.sock
mongo_1  | 2020-05-15T16:21:37.422+0000 I  NETWORK  [initandlisten] Listening on 0.0.0.0
mongo_1  | 2020-05-15T16:21:37.423+0000 I  NETWORK  [initandlisten] waiting for connections on port 27017
mongo_1  | 2020-05-15T16:21:37.423+0000 I  SHARDING [LogicalSessionCacheReap] Marking collection config.transactions as
collection version: <unsharded>
mongo_1  | 2020-05-15T16:21:38.588+0000 I  NETWORK  [listener] connection accepted from 172.21.0.3:44200 #1 (1 connectio
n now open)
mongo_1  | 2020-05-15T16:21:38.601+0000 I  NETWORK  [conn1] received client metadata from 172.21.0.3:44200 conn1: { driv
er: { name: "PyMongo", version: "3.10.1" }, os: { type: "Linux", name: "Linux", architecture: "x86_64", version: "4.9.13
-moby" }, platform: "CPython 3.6.3.final.0" }
web_1    | DEBUG:web_app:Initialising server...
web_1    | host name: server2_web_1
web_1    |  * Serving Flask app "app" (lazy loading)
web_1    |  * Environment: production
web_1    |    WARNING: This is a development server. Do not use it in a production deployment.
web_1    |    Use a production WSGI server instead.
web_1    |  * Debug mode: off
web_1    | INFO:werkzeug: * Running on http://server2_web_1:5000/ (Press CTRL+C to quit)
```

Figure 3.6: Each MPC party has their own interactive shell

Listing 3.7: Shamir secret sharing algorithm (part of MPyC framework)

```
def random_split(s, t, m):
    field = type(s[0])
    p = field.modulus
    order = field.order
    T = type(p) # T is int or gf2x.Polynomial
    n = len(s)
    shares = [[None] * n for _ in range(m)]
    for h in range(n):
        c = [secrets.randbelow(order) for _ in range(t)]
        # polynomial f(x) = s[h] + c[t-1] x + c[t-2] x^2 + ... +
            c[0] x^t
        for i in range(m):
            y = 0 if T is int else T(0)
            for c_j in c:
                y += c_j
                y *= i + 1
            shares[i][h] = (y + s[h].value) % p
    return shares
```

The algorithm above splits each secret given by parameter s (list of secrets) into m random Shamir shares. The degree for the Shamir polynomials is t remember $0 \leq t < n$. The algorithm returns a matrix of shares, one row per party.

Finally the user can start the computation by, sending a the appropriate request to one of the parties (listing 3.8). We make use of the requests package, requests is a simple HTTP library for python it enables us to make requests and get responses over HTTP.

Listing 3.8: Sending request for secure face matching task

```
# Define hosts and ports
hosts = ['localhost', 'localhost', 'localhost']
ports = [5000, 5001, 5002]

# Shares secrets
image = cv2.imeread("face.pgm")
kernel = [[1,1,1],[0,0,0],[-1,-1,-1]]
send_shares_mpc(image, ['Image'], 'test', hosts, ports, combined
     = True)
send_shares_mpc(kernel, ['Filters'], 'model', hosts, ports,
    combined = True)

# Start MPC
url = f'http://{hosts[0]}:{ports[0]}/mpyc_launch?api=
    face_matching_server'
response = requests.get(url)
# Response contains output of the MPC
```

Note that the HTTP response we get from the GET request contains two variables; the status code and the text. The status code indicates wheter the HTTP request has been succesfully completed, a valid and succesfull request has a response code of 200. The text of a response is the message that is transferred in the body of the HTTP response. We use the HTTP as means of communication between the device and the server. Note that for more secure implementations, HTTPS (Hypertext Transfer Protocol Secure) is preferred. Since traffic in HTTPS is encrypted using Transport Layer Security (TLS). Using HTTPS denies MITM attacks (man-in-the-middle), a type of attack where a passive attacker tries to eavesdrop certain packets of the traffic.

The GET query of the request sent in listing 3.8 executes the following code (listing 3.9) on the first party or PARTY_0 (localhost:5000).

Listing 3.9: Launching MPC and returning formatted output

```
is_running = False
@app.route("/mpyc_launch", methods=["GET"])
def mpyc_launch():

    def get_api_name(api_name):
        return api_name + '.py'

    http_arg = request.args.get('api')
    script_name = get_api_name(http_arg)
    if script_name is None:
        return "400"

    os.chdir(main_wd)
    test_path="./mpyc/demos"
    os.chdir(test_path)
    # Raise other parties
    Party = os.getenv(f"Party")
    # Only execute following code for PARTY_0
    if Party == '0':
        global is_running
        if is_running:
            return '200'
        else:
            is_running = True

        for i in range(int(os.getenv('N_PARTIES')) - 1, 0, -1):
            party_host = os.getenv(f'PARTY_{i}_HOST')
            party_port = os.getenv(f'PARTY_{i}_PORT')
            host_addr = f'http://{party_host}:{party_port}/
               mpyc_launch?api={http_arg}'
            r = requests.get(host_addr)
            time.sleep(2.50)

        # Run MPC script (PARTY_0)
        process = subprocess.Popen(['python', script_name, '-c',
            f'party{3}_0.ini'], stdout=subprocess.PIPE)
        stdout, stderr = process.communicate()
        is_running = False

        # Output everything inside of the '$$$' tags
        output_formatted = stdout.decode().split('$$$')[1]
        output_formatted = output_formatted.split('$$$')[0]
        output_formatted = output_formatted.strip()
        output_formatted = output_formatted.replace('\n', '')
        output_formatted = output_formatted.strip(',')
        return output_formatted
    else:
        # Run MPC script (party_1 and party_2)
        os.system(f'python {script_name} -c party{3}_{Party}.ini
            &')
        return "200"
```

The value of the api argument from the GET request determines which MPC script gets excecuted. Then PARTY_0 starts executing the same code snippet (mpyc_launch) on the other two parties, by sending the appropriate requests. The first party then executes a new subprocess that runs a the MPC script. The result of the MPC script gets written to the STDOUT stream (standard output) which can be read by the parent process. The result gets parsed in the MPC script, by printing the result between two $$$ tags (like this $$$ result $$$). The other two parties skip most of the code (because their environment variable 'Party' differs from 0) and go straight to executing the same MPC script. Notice that PARTY_0 plays an important role in establishing that the MPC script runs on the other parties. This centralisation is a clear weakness of this implementation, because a corrupted PARTY_0 could cause great harm to the correctness of the protocol. But for the sake of simplicity we opt for this more centralised implementation. An overview of this implementation with the different groups (device and server) can be seen in figure 3.7.

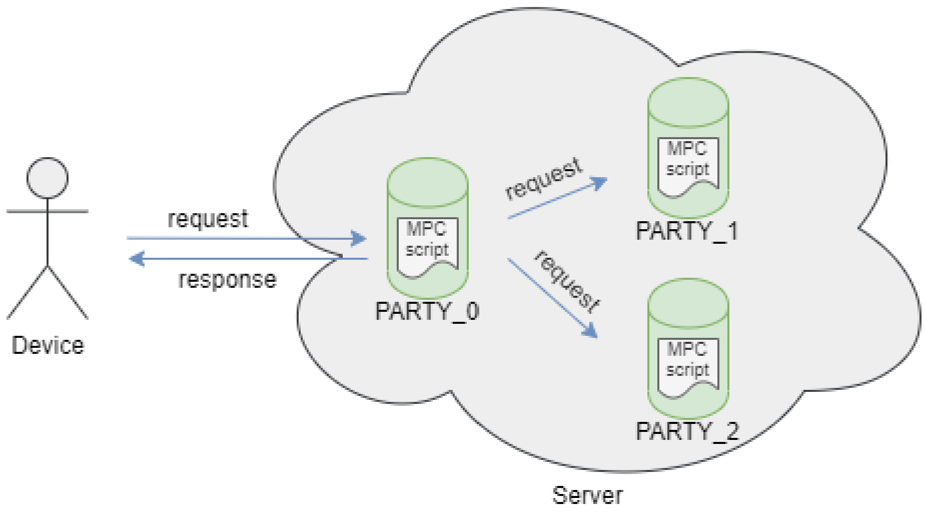


Figure 3.7: Overview of MPC implementation over HTTP (3 parties)

Last but not least, we discuss the mpc script that needs to run on the three parties. This script does the actual MPC. Each party holds an identical copy of this script. The parties begin by opening connections with each other (`mpc.start`). Remember: parties need to communicate with each other in order to output a result or to do non-linear operations such as multiplication of two secure numbers. Then the parties load the unique secret shares for the image and the convolution kernels stored in their MongoDB databases. Since the secret shares are stored in lists in the database, the parties reshape them to the original shape. Then the parties securely compute an output for the input image and kernels on a series of sequential layers (convolution, max pooling and ReLU) made out of the essential building blocks we introduced at the beginning of this chapter. Finally the parties output the result (mpc.output) print the result (printing in python has the same effect as writing to STDOUT) but they do this in a manner such that the stream is parsed. Note that all parties print the result, but only PARTY_0's STDOUT will be looked at. This is yet another flaw of our implementation. Better would be to look at everybody's STDOUT, to check if anyone is cheating by printing something other than the true result. Finally the parties begin shutting down (mpc.shutdown), closing all open connections. The subprocess will exit, unpausing the parent process. Which will unparse the STDOUT and return the result as a response to the device.

A basic example of an mpc script that contains all of the points made earlier can be observed in listing 3.10.

Listing 3.10: Example of MPC script for a single CNN layer (conv,maxp and relu)

```
import math
import numpy as np
from mpyc.runtime import mpc
from load_database import load_data
from custom_operations import convolution, relu, maxpool

async def main():
    await mpc.start()

    images = load_data('Image', 'test')
    kernels = load_data('Filters', 'model')
    image = images[0]
    kernel = kernels[0]

    image = np.reshape(image, (int(math.sqrt(len(image))), int(
       math.sqrt(len(image)))))
    kernel = np.reshape(kernel, (int(math.sqrt(len(kernel))),
       int(math.sqrt(len(kernel)))))

    conv = convolution(image, kernel)
    maxp = maxpool(conv)
    result = relu(maxp)
    result = list(np.asarray(result).flatten())

    print("$$$\n")
    print(await mpc.output(result))
    print("$$$")

if __name__ == '__main__':
    mpc.run(main())
```

Obviously a CNN has way more layers than just this triplet (convolution, max pooling and ReLU). But we will discuss the secure design of the CNN later on in chapter 3.2.2.

3.2 Design

In this chapter we discuss how we designed the neural network and why we chose for the network's architecture in specific. After explaining the neural network's architecture, we will have a look at how we translate a model working on plaintext pictures, to a secure model using MPC.

3.2.1 Convolutional Neural Network Architecture

In the process of choosing the best architecture for a CNN, we found four suitable models. We will see how each of these models are designed and in what way they differ from the others. Finally, we select one of these models to make it secure using MPC, the selection depends on the following criteria; straightforwardness of the MPC implementation, size of the network (number of parameters), accuracy of the model and the preferred output (distance vector or binary classification).

General (applies for all models): The input dimensions of the different neural networks is the same (1 x 100 x 100). The output can differ depending on what loss function is used (cross entropy loss or contrastive loss), but the meaning is the same, wheter a face matches another face or not. And the only difference is how to differentiate between these two classes (match or no match), this means the arbitrary threshold must be chosen on a different basis. To choose a good threshold that reflects the accuracy of the model, we iterate the calculation of the precision and recall for each threhold value in a certain range. This gets us the following curve (figure 3.8), out of which we can distract the best threshold.

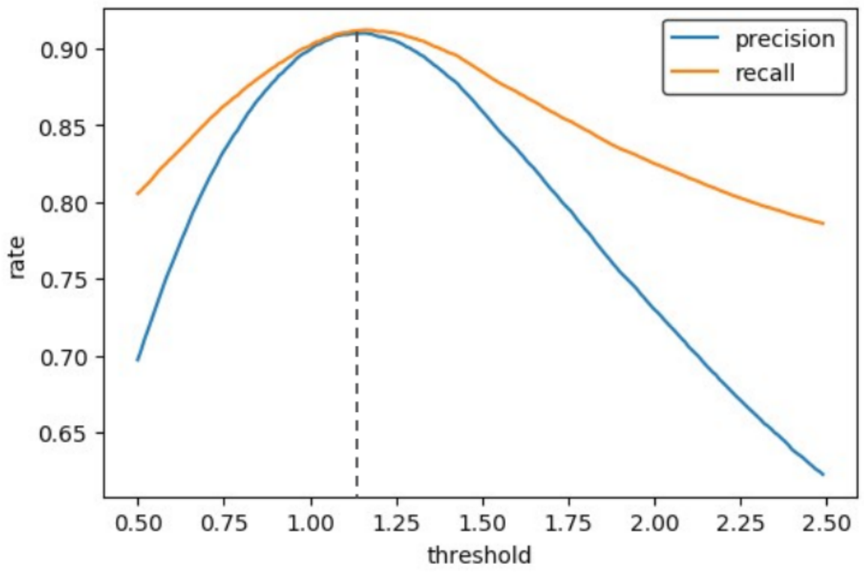


Figure 3.8: Curve for selecting the best threshold value

CNN with Contrastive Loss: This model is the first model we designed. It is simple and basic yet powerful. The neural network design can be seen in figure 3.9.

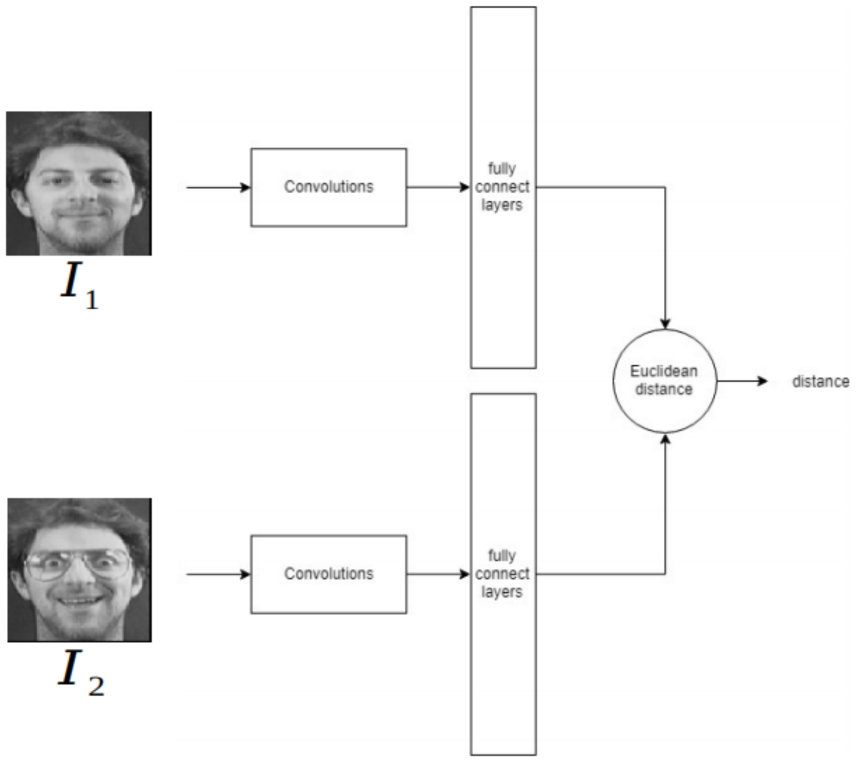


Figure 3.9: CNN with Contrastive Loss function

The convolution layers, including Max-pooling (2 x 2 with stride 2) and ReLU after every convolution, consist out of four layers:

1. 4 convolutions (7 x 7).
2. 8 convolutions (5 x 5).
3. 16 convolutions (3 x 3).
4. 32 convolutions (3 x 3).

Each of these layers have different tasks. The first layers have larger convolution filters, these convolutions detect high-level features like distance or rotation between ears, nose and/or eyes. While the latter, smaller convolution filters detect low-level features like lines or dots (jawline, eyes, mouth).

After the final convolution layer we flatten the output layer to a one-dimensional tensor. This tensor then serves as input for a small fully connected neural network (perceptron), this network can be seen as an extension of the CNN. We use this fully connected neural network to add global spatial correlation (a CNN is good for recognizing structures in local and neighbouring pixels). Since it's important to also important that the algorithm detects two objects or features that belong to each other even though they are seperated and are not local or neighbouring, think of two ears on a face for instance.

The total number of learnable parameters (weights and biases) in this neural network is 154.592, the model doesn't show any sign of overfitting nor underfitting up to 500 epochs.

The code for this neural network is printed in below (listing 3.11). Notice how simple this neural network is. With only four convolutional layers, it is able to make a face matching decision.

Listing 3.11: Code for CNN with Contrastive Loss

```
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()

        self.cnn1 = nn.Sequential(
            nn.Conv2d(1, 4, kernel_size=7),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.ReLU(),
            nn.Conv2d(4, 8, kernel_size=5),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.ReLU(),
        )

        self.fc1 = nn.Sequential(nn.Linear(512, 256), nn.ReLU(),
            nn.Linear(256, 64))

    def forward_once(self, x):
        output = self.cnn1(x)
        output = output.view(output.size()[0], -1)
        output = self.fc1(output)
        return output

    def forward(self, input1, input2):
        output1 = self.forward_once(input1)
        output2 = self.forward_once(input2)
        return output1, output2
```

CNN with Cross Entropy Loss: The following model (figure 3.10) is similar to the first one we described earlier. The key differences of this model, is that the siamese neural network stops after the last layer. The two output layers get flattened to two single one-dimensional tensors. These tensors then get concatenated. The concatenated tensor now gets sent trough a single fully connected neural network (perceptron).

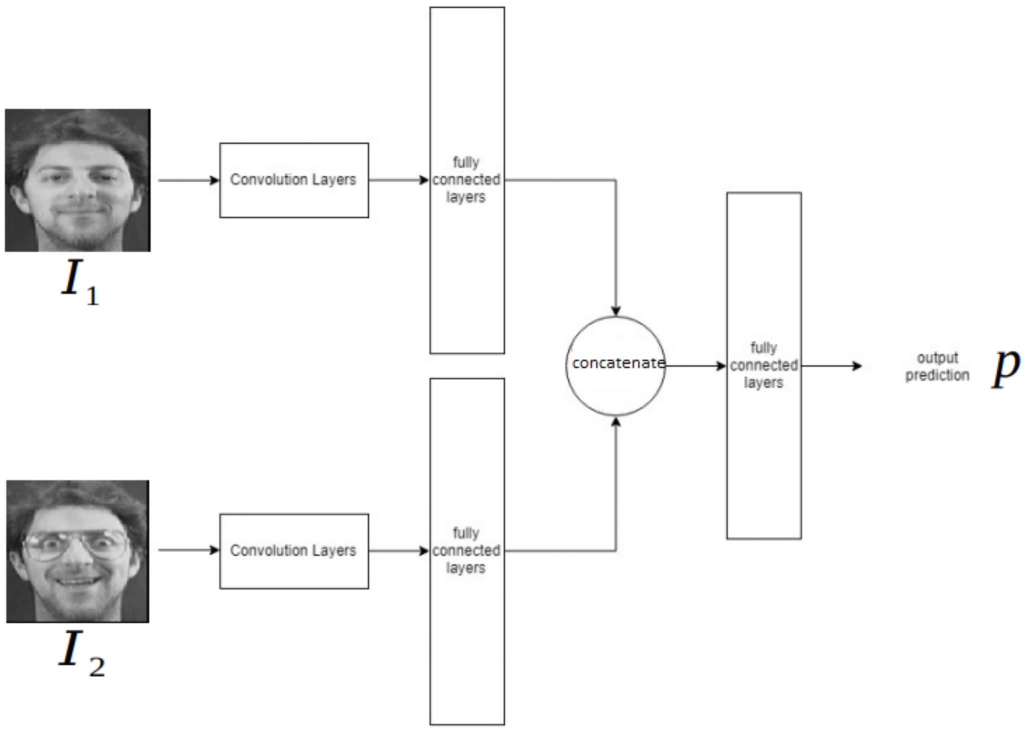


Figure 3.10: CNN with Cross Entropy Loss function

Note that in the first model (CNN with Contrastive Loss) there is no fusion of the siamese neural network. And the only time the two inputs depend on each other, is when we calculate the euclidean distance between the two output tensors. In fact, this type of model is not really a pure siamese neural network anymore. We guess that the perceptron, placed after the convolution layers, performs some sort of comparison on the two output tensors and we hope this comparison will be better than the standard euclidean distance.

Finally, the cross entropy loss is calculated on the last layer of the model. Since face matching is a binary task (matching or not matching), the last layer contains a single neuron (class). The cross entropy of that neuron returns the chance for that class to be true. The pleasant advantage of cross entropy is that we don't have to set up an arbitrary threshold value, the threshold is simply already decided, for instance if $p \in [0,1]$ than the threshold is equal to $p = 0.5$. If two faces are simmilar, $p = 0$. If two faces are dissimilar, $p = 1$.

The convolution layers consist of the same number of convolutions, as the first model:

1. 4 convolutions (7 x 7).
2. 8 convolutions (5 x 5).
3. 16 convolutions (3 x 3).

4. 32 convolutions (3 x 3).

The code for the model is listed below (listing 3.12)

Listing 3.12: Code for CNN with Cross Entropy Loss

```
class SiameseNetworkConcat(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()

        self.cnn1 = nn.Sequential(
            nn.Conv2d(1, 4, kernel_size=7),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU(),
            nn.Conv2d(4, 8, kernel_size=5),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=3),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU(),
        )

        self.fc1 = nn.Sequential(
            nn.Linear(1600, 100), nn.ReLU(inplace=True), nn.
                Linear(100, 10),
        )

        self.fc2 = nn.Sequential(
            nn.Linear(20, 20), nn.ReLU(inplace=True), nn.Linear
                (20, 1),
        )

    def forward_once(self, x):
        output = self.cnn1(x)
        output = output.view(output.size()[0], -1)
        output = self.fc1(output)
        return output

    def forward(self, input1, input2):
        output1 = F.relu(self.forward_once(input1))
        output2 = F.relu(self.forward_once(input2))
        output = torch.cat((output1, output2), dim=1)
        output = self.fc2(output)
        return output
```

Residual Neural Network: The last model makes use of a special type of deep learning, known as deep residual learning, introduced in 2015 by He et al. (2016). A Residual Neural Network (ResNet) is a special type of neural network that utilizes shortcuts or skip connections shortcuts to jump over a number of layers. Neural networks implementing residual blocks (figure 3.11) can have way more layers without running in to problems like vanishing gradient. Vanishing gradient problem occurs when certain parameters of a network do not get updated during training, because the gradient will be so small that it will prevent the parameters from changing. With ResNet a direct link (identy function) can be found over multiple layers, thus creating a way to link certain inputs deeper into the network.

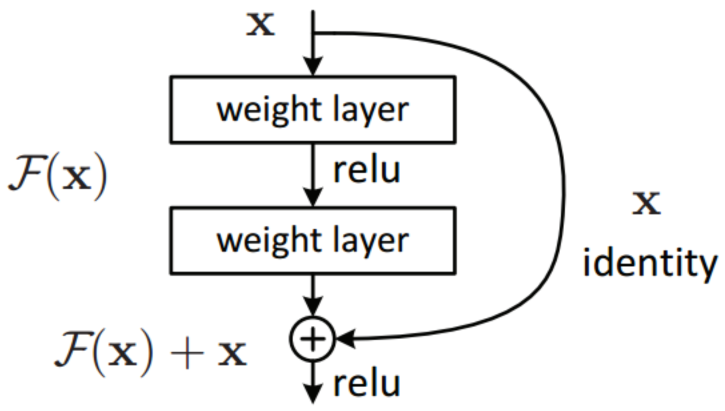

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 3.11: Residual block (2-layer skipping)

In short, ResNet allows us to use way more layers than before. So we derived our own ResNet-like model, from the pre-trained ResNet models available from Pytorch [6]. We saw an incline in accuracy using ResNet, but with more layers, there are more parameters and the complexity rises.

A basic residual block can be coded in Pytorch in the following way (3.13)

Listing 3.13: Pytorch Code for Residual Block

```
class BasicBlock(nn.Module):
    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.expansion = 1
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3)
        self.bn1 = nn.BatchNorm2d(planes)
```

[6] https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

```
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3)
        self.bn2 = nn.BatchNorm2d(planes)
        self.pool = nn.MaxPool2d(2,2)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes,
                   kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.dropout(F.relu(self.bn1(self.conv1(x))), p=0.3)
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

Notice that the `self.shortcut` function is simply the identity function.

As we said before choosing a neural network depends on a number of factors, we will now rank each of the neural networks described above for each determining factor.

Starting with the straightforwardness of the MPC implementation, in other words, how easy will it be to translate the code written with Pytorch to a MPC algorithm using no libraries for neural networks (since Pytorch doesn't support secure datatypes). It's obvious that a ResNet architecture requires more attention to detail and is more complex to implement. The other two neural networks require less attention to detail, thus are more easy to implement.

Next, we compare size of the neural network. The neural network's size or it's number of parameters is an equally important factor. Since MPC protocols are way slower than classic computations. We would like our neural network to do fast computations. Fast computations corresponds to less computations, and less computations corresponds to less (or smaller) convolutions or fully connected layers. Thus we would like our network to have a small number of layers, but still enough to be able to differentiate between faces. Since ResNet only adds extra layers, this model is again not preffered. The other two models have about the same number of parameters, and only the fully connected layers have a different number of weights and biases.

Ranking the neural networks on accuracy (the ability to match faces) was done by testing the four different networks, the results can be seen in table 3.1. We find that when we are using the

Contrastive Loss function, the classic CNN as well as the ResNet model is giving us better results than with the Cross Entropy loss function. Overall the ResNet model has a slightly better accuracy than the classic CNN.

Model	Loss	Accuracy
Classic CNN	Contrastive Loss	89.20%
Classic CNN	Cross Entropy Loss	81.94%
ResNet (8 layers)	Contrastive	91.06%
ResNet (8 layers)	Cross Entropy Loss	84.13%

Table 3.1 Accuracy of the different models

Last but not least, what output do we prefer? The models with Contrastive loss have an output in the form of a score, the distance between two output vectors gets calculated. While the models with Cross Entropy loss simply output a chance, not having to worry about choosing a certain threshold. During the testing of the networks we argued that determining the threshold is not too hard, in fact the threshold can be easily chosen by drawing out a graph of the accuracy of the network and the different thresholds as in figure 3.9.

Taking every factor in to consideration, we concluded that a classic CNN with Contrastive Loss offers a high enough accuracy and low complexity, while still being easy to implement in MPC.

3.2.2 Secure CNN

In the following part of this chapter we will discuss specifically what parts of the classic CNN we will encrypt. We ask ourselves the question: After how many convolution layers, ReLU layers and max pooling layers will the output be unrecognizable?

Suppose that the activations after the first n layers can not be traced back to the face present on the input image, then we don't need to encrypt the whole CNN but only the first n layers. We could split up the neural network in two parts; a part that is encrypted and uses MPC to do certain operations and a part that is unencrypted. Of course the encrypted part must be placed before the unencrypted part. Otherwise the face would not be encrypted.

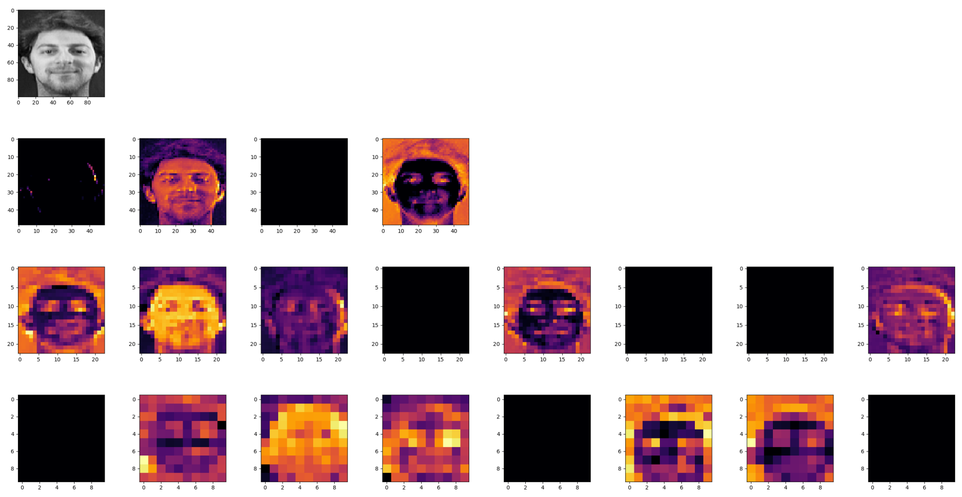


Figure 3.12: All Activation maps for an image

An activation map or feature map is the hidden output activations for a given convolution filter. They are called activation/feature maps because you can clearly see features or certain areas in the output that are more important than others (high activations). Activation maps give an insight to what features the CNN extracts from the input. In figure 3.12 an overview of all the activation maps for a certain image can be seen. The first image is the input. The four pictures below that are activation maps of the first layer. The eight pictures below those are the activation maps from the second layer. And the last eight pictures are the activation maps from the last layer.

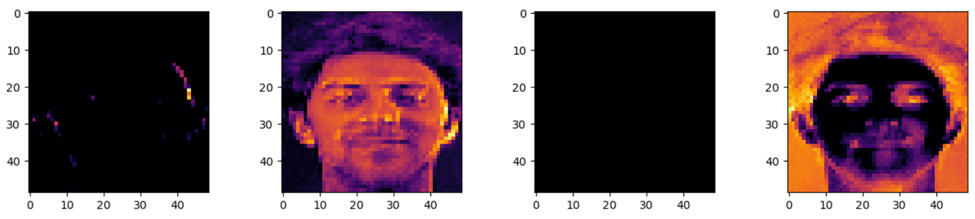


Figure 3.13: Activation maps after the first layer

As you can see in figure 3.13 the activation maps after the first layer are not obfuscated enough and can be recognized fairly easily by machines or humans. Thus only encrypting this part of the CNN would not be sufficient for a privacy preserving face matching algorithm. Notice how the resolution of the image became twice as small (this effect is caused by max pooling) and is thereby much more difficult to recognize. Another perturbation that is taking place is the ReLU activation function, discarding all negative values from the output. Lastly the convolutions can cause small to great perturbations to the image depending on the convolution filter.

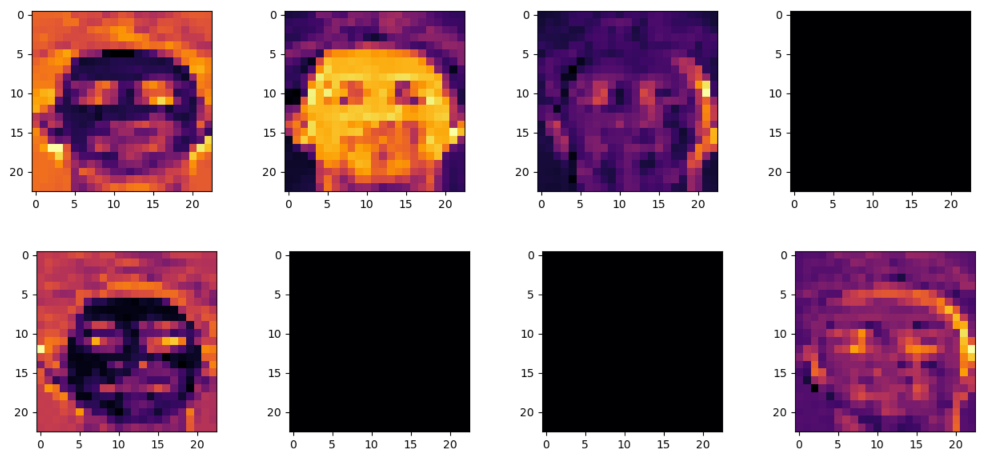


Figure 3.14: Activation maps after the second layer

The activation maps after the second layer (figure 3.14) are very pixelated. But since we can still recognize some major facial features, like the hair and the shape of the face. So this part of the neural network should still be encrypted.

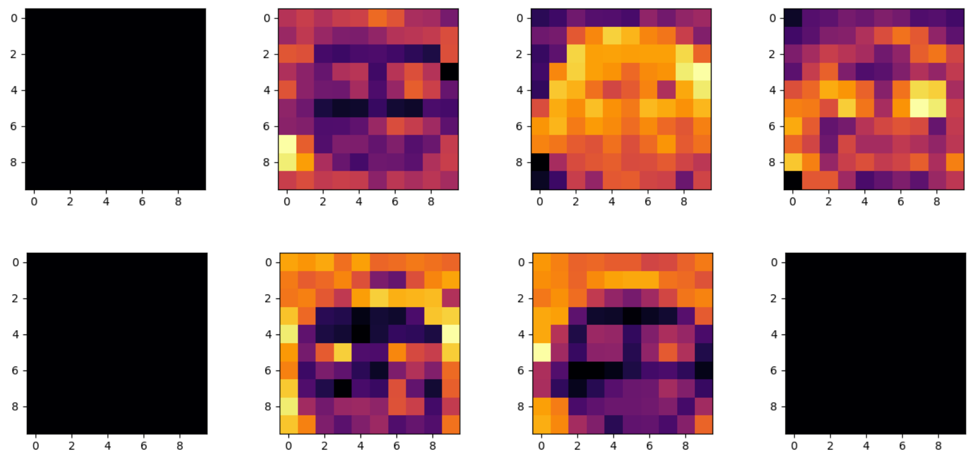


Figure 3.15: Activation maps after the third layer

Finally, we observe that it is impossible to recognize a face in one of the activation maps after the third layer (figure 3.15). The activation map is so pixelated that we can not recognize anything in it. Therefore we could make the third layer and everything that comes after it part of the unencrypted neural network. So that only the first two layers of the CNN needs to be encrypted using MPC. Of course for additional security one may choose to encrypt more than only the first two layers or even the whole neural network. But keep in mind that opting for more security by encrypting more layers, comes with an additional cost in complexity. We call this the security-complexity tradeoff.

The attentive reader may have noticed that certain of these activation maps are entirely black or predominantly black, if not have a look at figure 3.16, meaning they don't contain any information or are useless. We realized that this phenomenon was occuring on the same activation map for every single image which inferred through the CNN. The convolution filter's weights were close to zero or negative, this meant that any input to those specific convolutions would have an output close to zero or negative. In fact this is not really a problem but after the ReLU activation layer all of the values of the activation map would be equal to zero. Rendering the activation map and the convolution as a whole obsolete. To take an advantage of this we used some form of pruning.

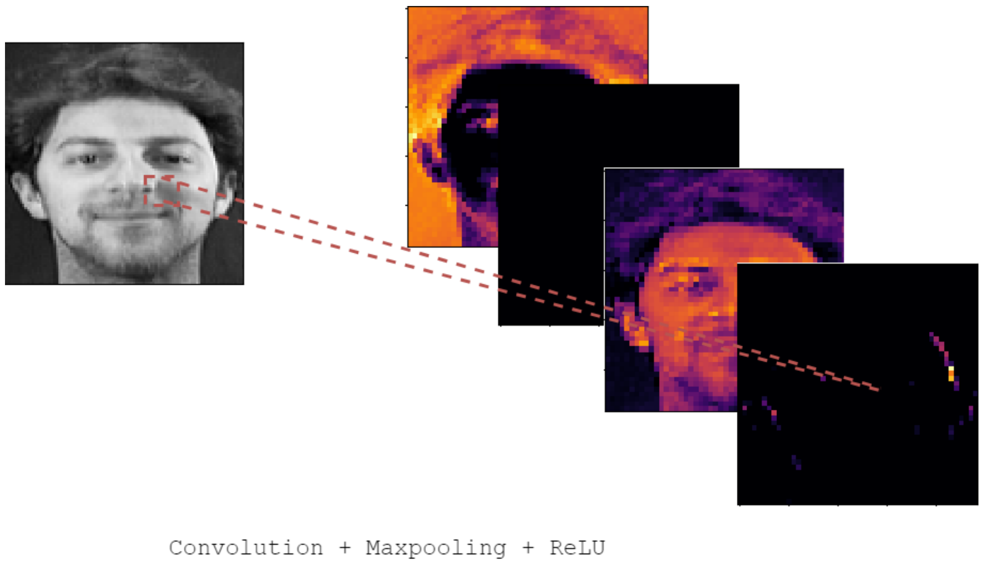


Figure 3.16: First layer without pruning

Pruning is a technique in machine learning that enables neural networks to be more efficient. It's an optimization technique that cuts out any unnecessary weights or biases. The result is a compressed, smaller version of the neural network than before, making it faster and less complex. There exists different processes and pipelines for pruning, and often neural networks are given too much parameters on purpose to apply pruning later on.

In our case we can safely remove any of the convolutions causing the activation map to turn out black. And without losing too much accuracy we can also remove any convolution filter that doesn't provide enough information and is deemed insufficient. The following figure 3.17 displays how the first layer only outputs two activation maps after pruning, instead of the four activation maps that were produced in the original CNN.

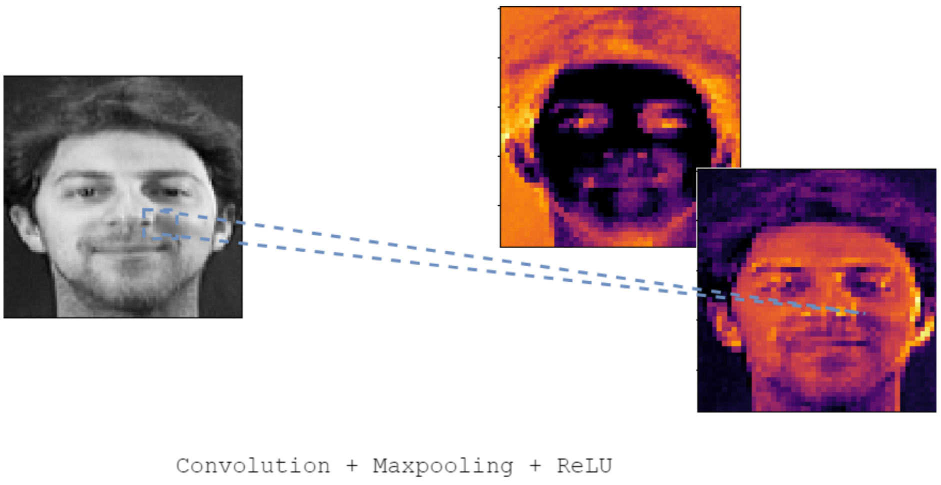


Figure 3.17: First layer after pruning

Eight out of the twenty convolution filters are infected with this defect, thus can be ignored or even removed from the CNN. This is great news, since this means we are able to optimize the convolutional part of the CNN with a speedup of around 40%.

Now it is really important that we do the same exact operations that lead the same unencrypted/encrypted input to the same unencrypted/encrypted output. To be sure of how each and every operation works in the cleartext version of the face matching algorithm we strictly follow the Pytorch documentation[7]. The `torch.nn.Conv2d(in_channels, out_channels, kernel_size)` method applies a 2D convolution over an input composed of several input planes. This means that it is possible that more than one tensors can serve as input. This is true in our case for each convolution layer except the first one. The method is implemented as equation 3.4 where C denotes the number of channels, $\star$ is the 2D cross-correlation operator, H is the height of the input planes and W is the width.

$$out(C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(k) \tag{3.4}$$

Notice that in our secure implementation of the `torch.nn.Conv2d(in_channels, out_channels, kernel_size)` we forgot to add biases and we only allow for one input channel. In fact our implementation looks more like a simple cross-correlation operation.

Adding the biases is really simple. And in fact we can just add the bias to each element of the output by defining a new method `add_bias(tensor, bias)` (listing 3.14). This method takes two argument; a 2D tensor and a single floating-point bias.

Listing 3.14: Code for adding bias to matrix

```
def add_bias(tensor, bias):
```

[7]https://pytorch.org/docs/stable/nn.html

```
    return np.add(tensor, bias)
```

The only difference that remains, is that the secure convolution function must allow a combination of input planes. To accomodate for this feature, we define a new method `conv2d(input, kernel, bias)` that computes the exact same output tensor as the vanilla Pytorch 2D convolution function with only one difference, it being fully encrypted. Notice that we make use of earlier defined methods.

Listing 3.15: Code for computing total 2D convolution

```
def conv2d(inputs, kernel, bias):
    output = convolution(inputs[0], kernel)
    for i in range(1, len(inputs)):
        output += convolution(inputs[i], kernel)
    add_bias(output, bias)
    return output
```

For the remaining parts of this thesis, when we talk about one convolution we imply the algorithm stated above (listing 3.15), except when clearly stated.

Of course the convolution needs to be done for each convolution filter on the apropriate set of inputs and in the correct order. Just as in the cleartext algorithm (listing 3.11), the convolutions are directly followed by the max pooling layers, which are in turn, followed by the ReLU activation layers. With the help of the three main building blocks of the secure CNN we can build convolutional part of the CNN. The other part, which entails the fully connected layers, doesn't need to be encrypted, since it is nearly impossible to recognize someone from a one dimensional tensor of activations, even with knowledge of the parameters of the neural network. Thus, this latter part can be computed on only one of the three servers or on the clients device. The calculation of the euclidean distance between the two output tensors must be done on the client's device, because this way the fingerprint (output tensor) of the validation face (the face of the image that we would like to match).

The secure face matching algorithm thus exists out of two parts. The first n encrypted layers make up the first part. The second part includes the rest of the convolutions for which the input tensors are deemed unrecognizable and the fully connected layers. The second part can be computed in the classical way on one of the three servers. MPC is only needed for the first part of the algorithm. This heavily reduces the complexity without compensating for security or privacy.

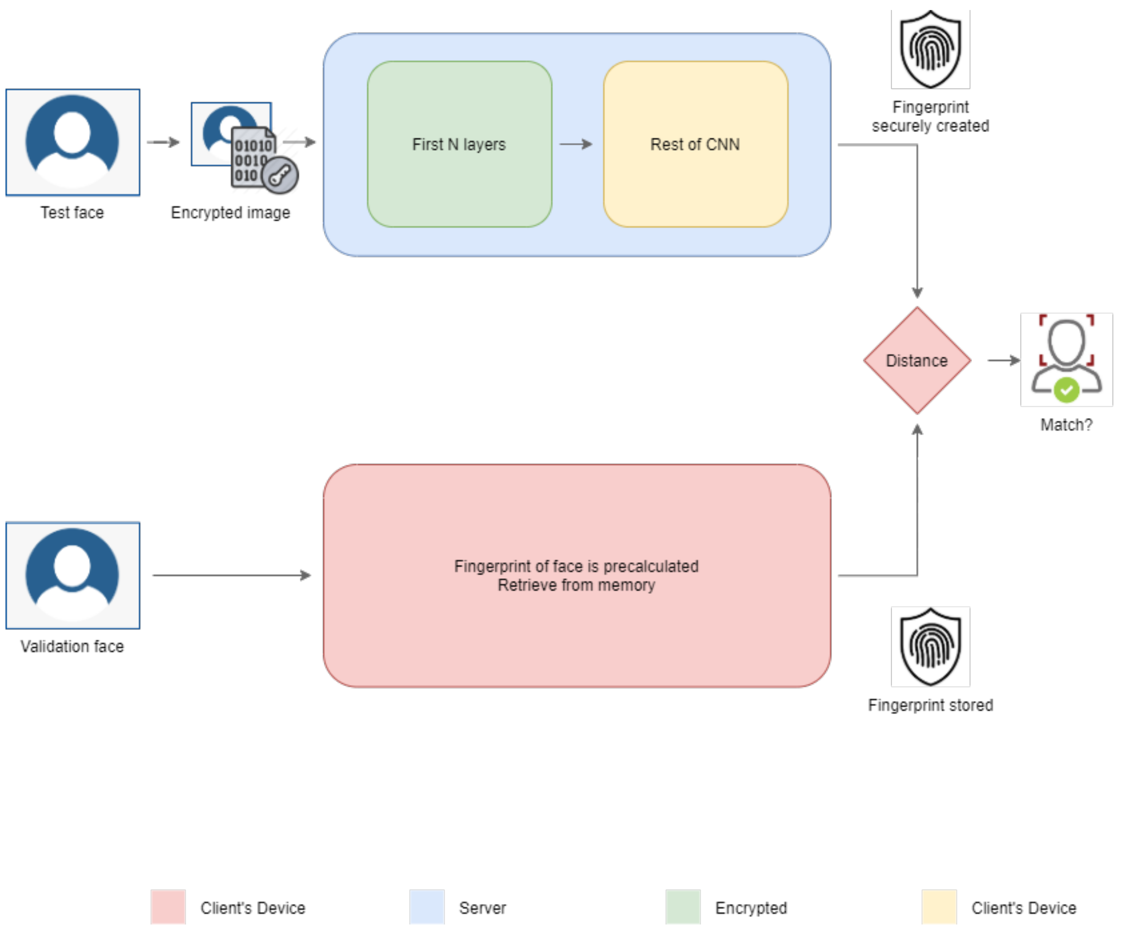


Figure 3.18: Overview of practical implementation of secure face matching algorithm

Our secure face matching algorithm still isn't complete. For a match to be succesfull a test image needs to be matched to a validation image (or base image). We could calculate the embedding (fingerprint) of this validation image for each face matching task by inferring the image through the CNN. But the result would always be the same, since the input doesn't change. Therefore it is sufficient to calculate the fingerprint of the image once and store it for a certain period on the client's device. If the fingerprint of the validation face is never shared and stays on the client's device during all times. Than it is safe to assume that the outcome of the face matching algorithm, a match or not a match, is also secure and private. An overview of this implementation is given in figure 3.18. The legend describes what parts of the process happen on the client's device or on the servers, and what parts are encrypted or public.

3.3 Conclusion

Implementing a secure face matching algorithm is not a straightforward process. This chapter describes what decisions we made, and what choices led to our implementation of the secure face matching algorithm. Of course there are several ways to implement such a complex task, but we will not be discussing those. A lot of effort has been put in optimizing the algorithm such that three core principles are obeyed:

1. The computation of the face matching task needs to be outsourced to public servers as much as possible.
2. The public servers can not recognize the input images nor the output of the face matching task.
3. The face matching algorithm must be reliable and accurate.

If and only if all three of above principles are verifiable, the face mathcing algorithm tends to give good results, is secure and the privacy of the client is protected.

4 Evaluation

In this chapter we will take a look at the results of various experiments conducted during the process of making a secure CNN for face matching. First we describe our testing environment in order to make reproducable results. Then we move on by explaining our first experiment, where we sharpened an image with the help of convolutions of course we made a secure version of this algorithm, which keeps the input image and the convolution filters parameters private. Then we will test the complexity of each of the different layers of the network; convolution layer, subsampling layer and activation layer. Finally we discuss our obtained results and talk about the problems we came across.

4.1 Testing Environment

Following is a list of hardware and software used during testing the results can vary and depend on following items.

Hardware

- Machine: Lenovo ThinkPad T460s
- CPU: Intel Core i5-6300U (2.4 GHz)
- Memory: 8 GB DDR4

Software

- Windows 10 Pro (device) and Linux (servers)
- Python version 3.6
- Virtualization: Docker Desktop
- Database: MongoDB (NoSQL)
- Python packages: MPyC and Pytorch

To ensure that the secure face matching algorithm works as intended, all software listed below should be installed and the hardware should meet similar specifications as the hardware we used.

4.2 Image Sharpening

Image sharpening is a classic computer vision task. The task can be completed using convolutions. The process goes as follows: First we extract the horizontal lines or edges from an image by using a certain convolution mask (figure 4.1). We do the same thing for the vertical lines in the image. Adding these outputs gives us a detailed version of the input image, where only vertical and horizontal lines are left over. The detailed image than gets multiplied by a factor between 0 and 1. Finally the detailed image gets added to the input image, the result is a sharpened version of the original image.

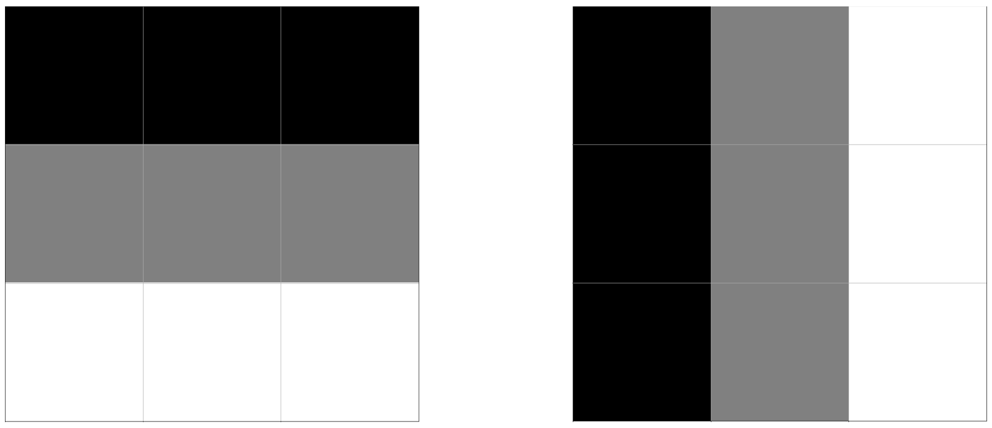


Figure 4.1: Convolution filters used for extracting sharp details out of images

This demonstration nicely illustrates the possibilities for using secure convolutions and possibly other image processing tasks such as corner or edge detection. There are a total of two convolutions filters sized $3x3$. For an input image of size $100x100$ the total MPC running time which includes the two secure convolutions, the multiplication of the factor and the addition of the detailed images to the original images, equals to 60-80 seconds. For a bigger picture (figure 4.2), sized $640x360$, the computation takes much longer since the number of convolutions goes up drastically. The MPC computation time for those bigger pictures can take up to 5 minutes. An example of a securely sharpened image can be found below.

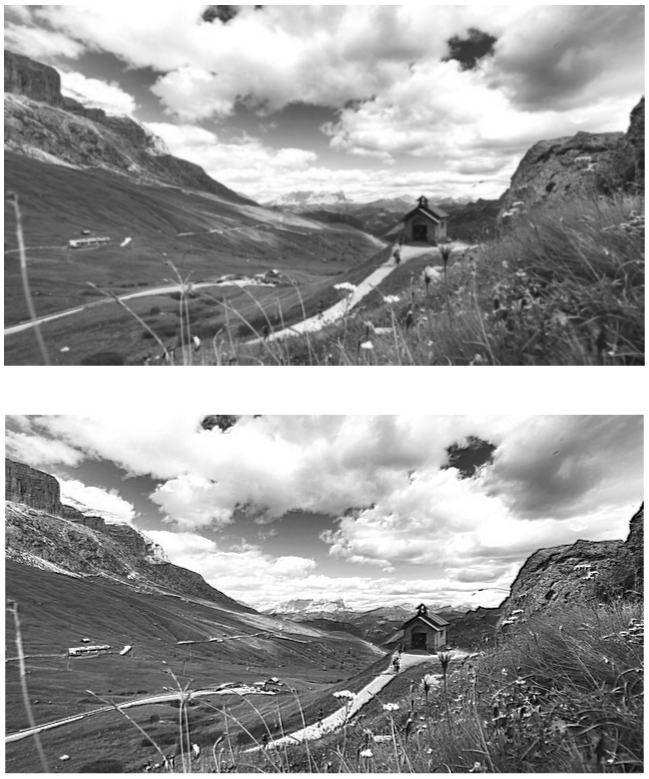


Figure 4.2: Convolution filters used for extracting sharp details out of images

Notice that the contours and sharp lines in the original image are much more visible in the sharpened image. Let us revise what exactly happend from a security point of view. The client takes a picture with their device. Then they encrypt the picture, by using Shamir's secret sharing scheme. The secret shares of the encrypted image then gets sent to the appropriate three MPC servers. These servers now do the convolution to sharpen the image. Note that the parameters of the convolution filters are also encrypted, this means that the intellectual property of the algorithm stays protected. Of course this is not really usefull for the image sharpening algorithm since it's publicly known what parameters are best chosen for the convolutions. But for algorithms that are not already known and would lose their value or income if they would be revealed, this feature would enable the parameters of those algorithms to stay private. After the server are done with the MPC protocol. The servers recombine their secret shares to get the output, the sharpened image.

4.3 Complexity results

In this section we show the complexity of the different secure operations (single convolution, max pooling and ReLU) and the different parts of the MPC protocol, secret sharing and share recombination for example. We do this by measuring how long it takes to perform the operation on a matrices with variable sizes. We then compare these results to their classic unencrypted equivalents.

We define our testing vector (the input image) as I of size sxs, the kernel K of size 3x3 and the computation time t. Note that we only measure the computation time of one party. However that is fine, since the parties are running parallelized and they take around the same time to finish their computations.

The testing vector I is randomly generated with values uniformly distributed over the range $[0,1]$.

4.3.1 Convolution

For this operation we have four different implementations as can be seen in figure 4.3. The first one uses MPC to encrypt the input I. The second one uses MPC to encrypt both the image I and the kernel K. The third convolution function is `torch.nn.Conv2d` from the Pytorch package. The last function is the cleartext equivalent of the MPC secure function but instead of using secret shares as input it only takes cleartext floating points and does not use the MPC framework.

The first two functions and the last function are defined in listing 3.2 of chapter 3.1.2.

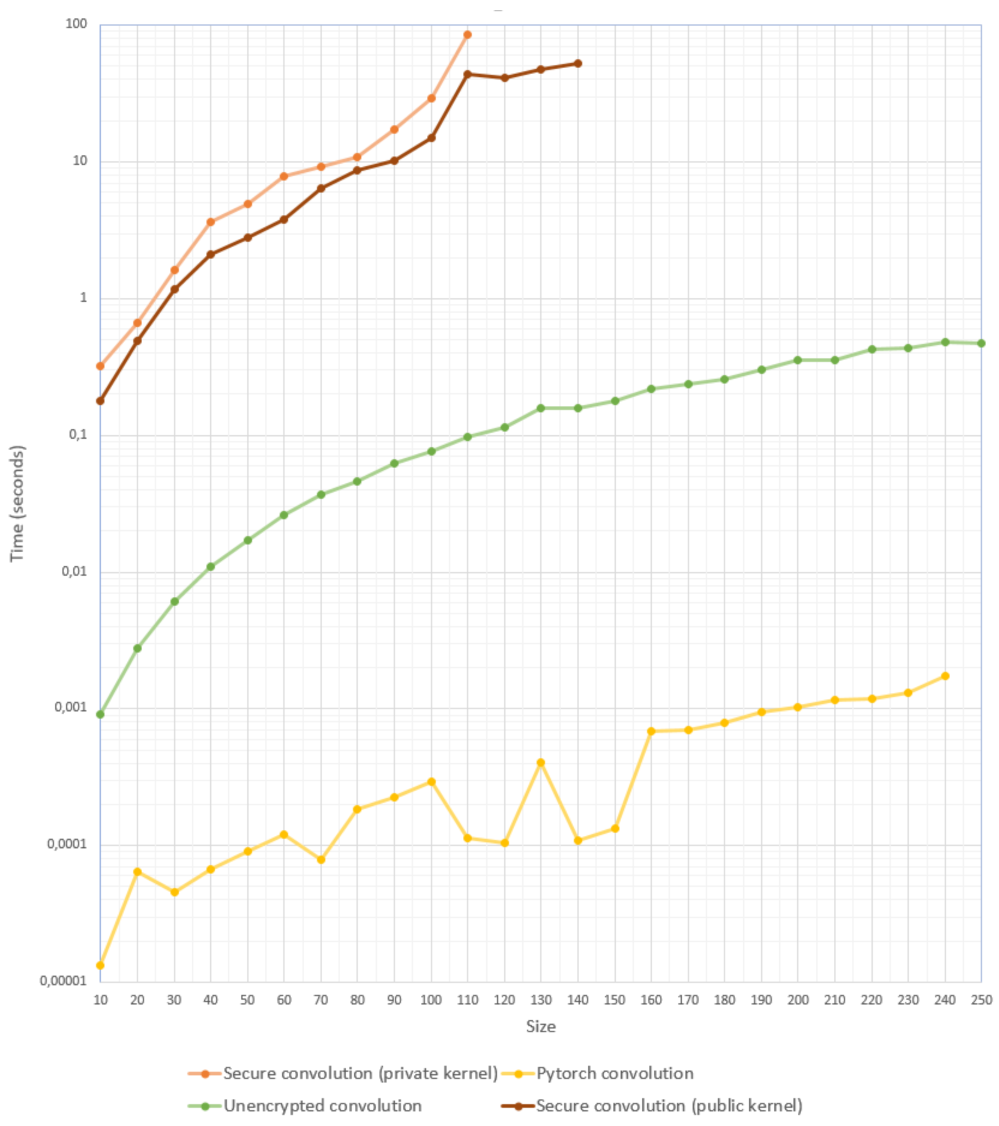


Figure 4.3: Computation time for encrypted and unencrypted convolution functions

We observe that the Pytorch function is the fastest one of the four. The unencrypted convolution function is of a magnitude 10^2 slower. The two other encrypted convolution take about the same time to complete. However, the convolution function where the kernel is also encrypted is a tad slower. This is logical, since communication between the MPC servers is needed for multiplication of two secrets (as seen in chapter 2.2.2.1).

4.3.2 Max Pooling

In figure 4.4 three different max pooling functions are plotted over the size s of the input I, while these functions are differently implemented the result is the same. The window size is 2 and the stride is also 2. The secure max pooling function uses MPC as means to encrypt the input I and is defined in listing 3.3 of chapter 3.1.2. The pytorch max pooling function is `torch.nn.MaxPool2d` of the Pytorch package. The unencrypted max pooling function is the equivalent of the first encrypted function but instead of secret shares it uses cleartext floating points as input, and it does not utilize the MPC framework at all.

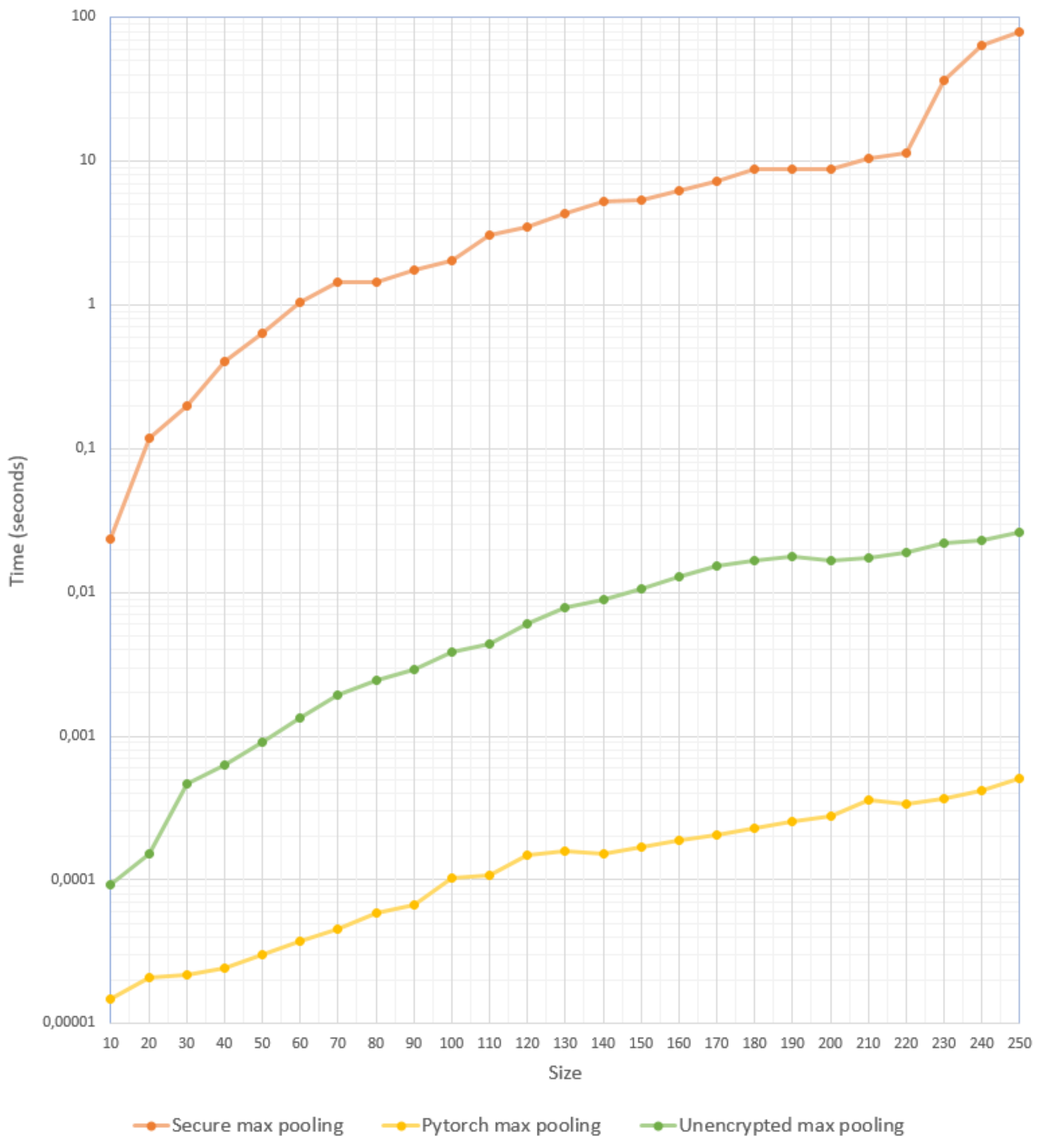


Figure 4.4: Computation time for encrypted and unencrypted max pooling functions

We can clearly see that the Pytorch function is the most optimized function and runs the fastest. The unencrypted max pooling function is a factor of 10^1 to 10^2 slower. The MPC encrypted max pooling function is the slowest and reaches almost 100 seconds for max pooling over an image I of size $s = 250$.

4.3.3 ReLU

In figure 4.5 three different functions are plotted for an image I with sizes s in range $s \in [10, 250]$. The secure ReLU function is defined in listing 3.4 of chapter 3.1.2 and uses MPC to encrypt the input I. The Pytorch ReLU function is `torch.nn.ReLU` from the Pytorch package. The results for this function shows very fast computation, this is because Pytorch is deeply integrated with C++ code. Finally, the unencrypted is the insecure equivalent of the first function but instead of using the MPC framework and giving secret shares as input, regular floating points are given as input.

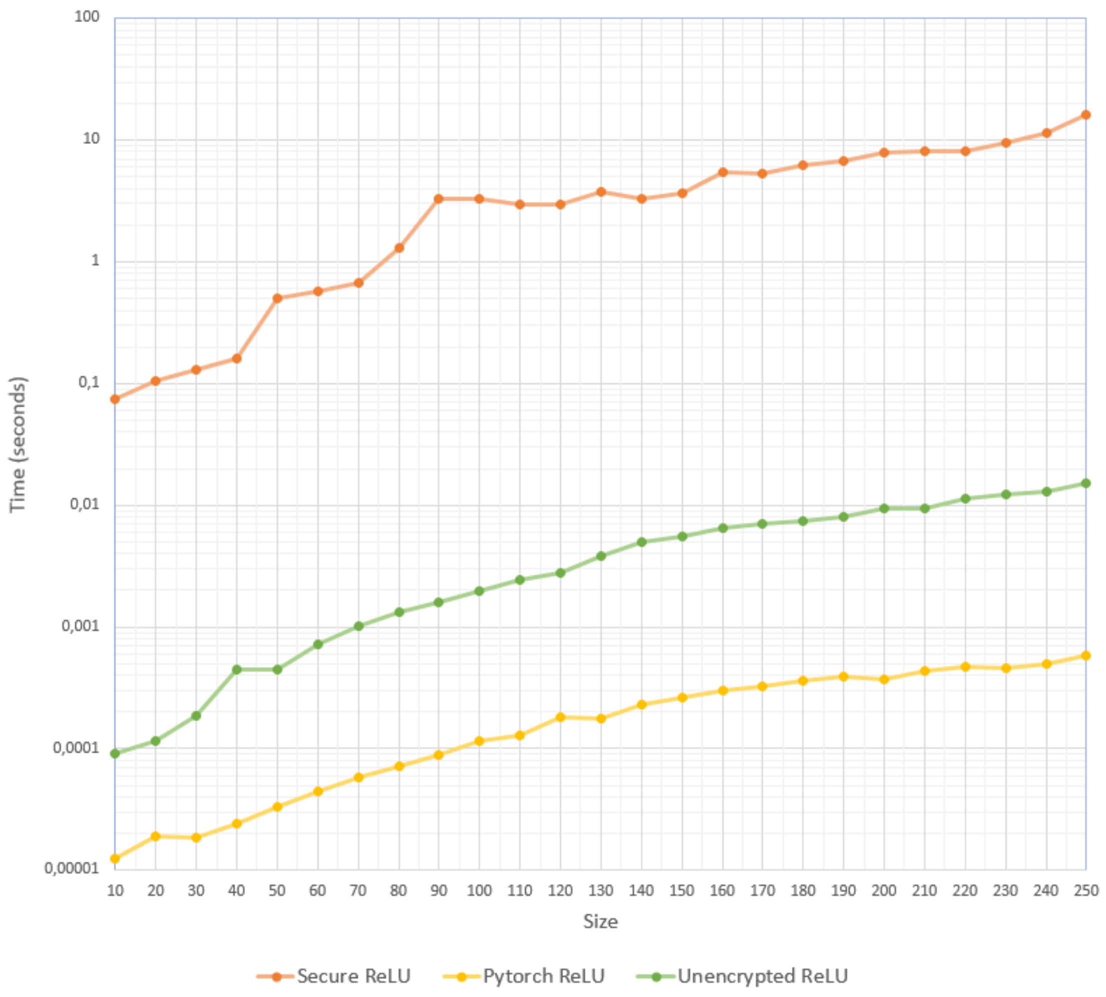


Figure 4.5: Computation time for encrypted and unencrypted ReLU activation functions

The results are predictable, of course the Pytorch function is the fastest because of its utilization of C++ code. The unencrypted ReLU function is about ten times slower. The MPC secure function is about 10^4 as slow as the Pytorch function and 10^3 as slow as its unencrypted equivalent.

4.3.4 Secret sharing and recombination

Complexity analysis of Shamir's secret sharing scheme and secret recombination via Lagrange interpolation (chapter 2.2.1) is less important than the other operations. Because they happen only once for every MPC instance, secret sharing happens at the beginning of the protocol while secret recombination takes place at the end. However they are essential parts of the MPC protocol. In our experiment (figure 4.6 we let the computation of secret sharing happen on the device and secret recombination on the MPC servers. The time for those operations is measured for and image I with size s ranging from 10 to 250.

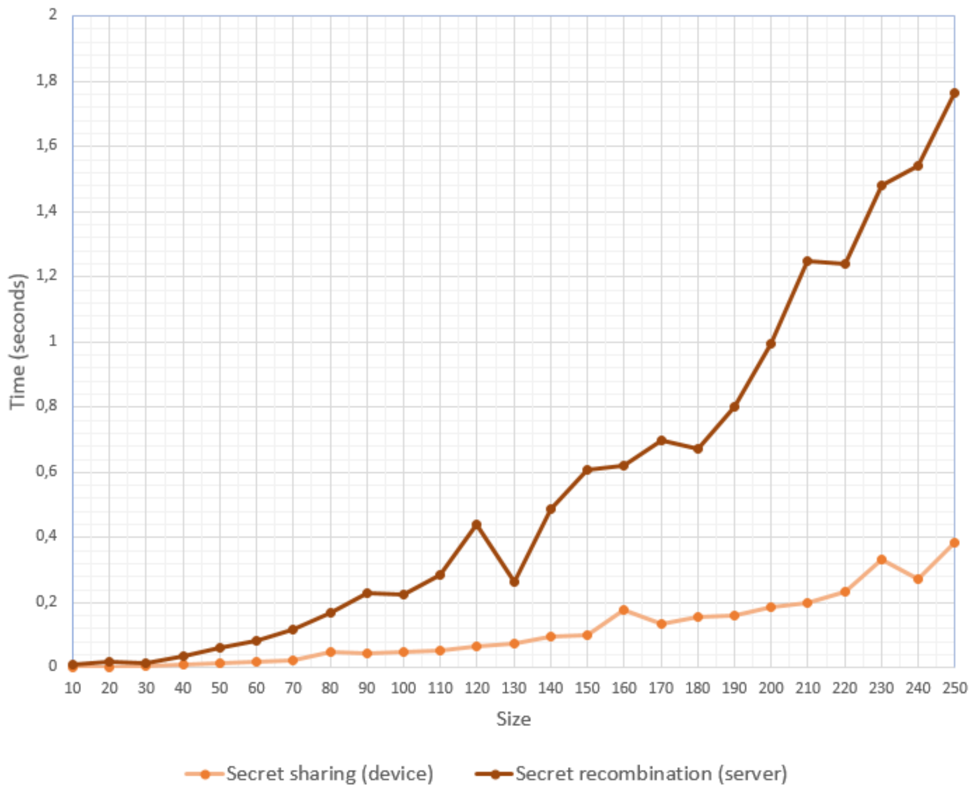


Figure 4.6: Computation time for secret sharing and recombination algorithms

Notice that secret recombination takes more time, this is normal since communication between the parties is needed to publish the secret shares. Overall these operations are relatively fast (comparing to the other secure operations) and since they only occur once for every secure face matching task, their computation time is to be disregarded.

4.4 Discussion & Problems

We had difficulties implementing the whole CNN in the MPC framework. To a point where it seemed impossible to continue. The docker containers running the MPC parties instances, crashed as soon as the computation began to be too complex. We couldn't figure out how to run multiple convolution layers after eachother without crashing the system. We still have not figured out why a large input or a complex operation caused the docker containers to crash. Because of how difficult it is to backtrack the problem. We do not have enough experience regarding Docker. And since the goal of this thesis is not to master our debugging skills, we are sad to announce that we could not get the total secure face matching algorithm to work. However we have made significant progress by designing and testing the different secure methods needed for this face matching algorithm.

In theory our implementation is entirely feasible. But in practice more than just a couple of bright theoretical ideas are needed. For starters, a proper software stack that is indifferent to the size of the input or to the number of operations is much appreciated.

The crashes on the servers cause a connection timeout on the device. And happen in the following two cases (causes gathered from experience):

- The number of input pixels is greater than 130x130 for a single convolution.
- The number of convolutions for an input of size 100x100 is greater than one.

Another problem is the fact that the secure operations are very complex and take a lot of time to complete. An estimation of the total time it would take to complete one inference of the secure CNN in it's current form (with pruning) is calculated below (equation 4.1) taking into account the results from the computation time tests.

Secure face matching algorithm step per step:

- 1 x secret sharing on I with $s = 100$
- 2 x single convolution on I with $s = 100$
- 2 x max pooling on I with $s = 98$
- 2 x ReLU on I with $s = 49$
- 10 x single convolution on I with $s = 49$
- 5 x max pooling on I with $s = 47$
- 5 x ReLU on I with $s = 23$
- 25 x single convolution on I with $s = 23$
- 5 x max pooling on I with $s = 21$
- 5 x ReLU on I with $s = 10$
- 5 x secret recombination on I with $s = 10$

$$\begin{aligned}
t_{total} =& 1 \times 0.0476166 + 2 \times 14.8732678 + 2 \times 2.0341087 + 2 \times 0.4950582 \\
& + 10 \times 2.792582 + 5 \times 0.6305378 + 5 \times 0.1049295 + 25 \times 0.4898818 \\
& + 5 \times 0.1172477 + 5 \times 0.073816 + 5 \times 0.0109198 \\
t_{total} =& 79.712605
\end{aligned} \tag{4.1}$$

The total computation time for one inference through the secure CNN is about 80 seconds. But this seems a bit contradictive and we are sure some elements like parties waiting on eachother are not taken into account. But since we do not have a practical working implementation of the secure face matching algorithm we can not say for certain how long it would take to complete one task.

5 Conclusion

In this chapter we will provide and answer to our initial hypotheses from section 1.2 with our knowledge of the literature study from chapter 2 and the results of chapter 4. We will also provide hints to continue improving our work and in which directions we would look for future studies on this subject.

How can we securely compute the inference of a deep learning-based facial recognition neural network?
Secure multiparty computation (MPC) is a cryptographic protocol which ensures that certain operations can be done on private inputs without revealing any information about the input to the entities doing the outsourced computing. MPC heavily depends on older concepts such as Shamir's secret sharing scheme for encryption and Lagrange interpolation for decryption. The encryption/decryption is not based on (a)symmetric-key cryptography instead it achieves its security through the fact that all parties must be corrupt in order to decrypt the encrypted data. Since convolutional neural networks (CNN) are simply put, extremely large non-linear functions, we have been able to adapt certain functions that are used in CNNs so that they can be used with MPC.
The MPC protocol needs at least three independent parties (preferably more). These parties do secure operations on encrypted data to output the correct result (as it would have been if the computation was done without MPC). We will discuss four cases that can occur. Firstly, the parties can behave honest but curious. The privacy of the input will be preserved and the result will be correct. Secondly, the majority of parties have a malicious intent and will try to alter the result of the computation but atleast one party is honest and will not collude with the other parties. The privacy of the input will be preserved but the result of the output could be wrong and biased in a way that suits the needs of the majority. Finaly, all parties are corrupt through bribery or directly through interest

in the encrypted data. It will be possible to decrypt the private input and all parties participating in the MPC protocol will be able to see the decrypted input. This is the worst case scenario, however this state of full-scale corruption is incredibly hard to get. Since all it takes to prevent this, is only one honest party.
The drawback of encrypting the inference of a CNN using MPC is that secure MPC operations take much more time to compute. Every party needs to do the same amount of work and some operations require communication between the partiesn, these latter operations make the total computation time of one inference very slow. Since the parties are only as strong as the weakest link in the chain.

How can we optimize the secure facial recognition task to run more efficiently?
Since there are limits on the optimization of underlying protocols and systems such as the latency and bandwith of a network and the clock frequency of CPU,... . Other things should be considered before upgrading these existing protocols and systems. First of all, the whole CNN should not be securely computed. Only the parts where the input and output tensors are recognizable should be encrypted. After a couple of neural network layers the data will not be comparable to the input. Secondly the fully connected layers operate on unrecognizable data and thus should stay unencrypted as well. Finally, certain convolution layers are not doing anything usefull and will simply output an all-zero tensor. These convolutions can be omitted from the secure inference in order to gain efficency without losing any accuracy.

5.1 Future Work

We have arrived at the last section of this thesis. However, our work doesn't end here. This project offered an insight as to how a theoretical as well as a practical implementation of a privacy-preserving, deep learning-based, facial matching algorithm using secure multiparty computation as means of encryption, works and how reliable the protocol is. Since we were not able to complete a working proof of concept, we strongly emphasize anyone working on this study or studies related to privacy-preserving machine learning to learn a thing or two from our mishaps.

There are several improvements that can be done. As for one, the MPyC integration with Pytorch could easily be added by allowing `torch.tensors` to work in MPyC, instead we had to copy the values from the tensors to arrays. Secondly, it would be interesting to compare our timing results with timing results from the same MPC operations on physical dedicated servers[1] that are geographically seperated, to mimic real-life implementations of MPC. Finally, it would be interesting to demonstrate the different types of attacks on the protocol discussed in chapter 2.2 and search for ways to prevent them.

[1] Amazon Web Services or Digital Ocean droplets

Bibliography

Asharov, G. and Lindell, Y. (2017). A full proof of the bgw protocol for perfectly secure multiparty computation. *Journal of Cryptology*, 30(1):58–151.

Barni, M., Orlandi, C., and Piva, A. (2006). A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pages 146–151. ACM.

Ben-Or, M., Goldwasser, S., and Wigderson, A. (1988). Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM.

Cadwalladr, C. and Graham-Harrison, E. (2018). Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The guardian*, 17:22.

Campmans, H. (2018). *Optimizing Convolutional Neural Networks in Multi-Party Computation*. PhD thesis, Master's thesis, Dept of Mathematics and Computer Science, TU Eindhoven.

de Hoogh, S. J. A. (2012). Design of large scale applications of secure multiparty computation: secure linear programming.

Erkin, Z., Franz, M., Guajardo, J., Katzenbeisser, S., Lagendijk, I., and Toft, T. (2009). Privacy-preserving face recognition. In *International symposium on privacy enhancing technologies symposium*, pages 235–253. Springer.

Franklin, M. and Haber, S. (1996). Joint encryption and message-efficient secure computation. *Journal of Cryptology*, 9(4):217–232.

Gentry, C. et al. (2009). Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178.

Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. (2016). Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210.

Hadsell, R., Chopra, S., and LeCun, Y. (2006). Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Koch, G., Zemel, R., and Salakhutdinov, R. (2015). Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2.

Mainali, P. and Shepherd, C. (2019). Privacy-enhancing fall detection from remote sensor data using multi-party computation. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, page 73. ACM.

Makri, E., Rotaru, D., Smart, N. P., and Vercauteren, F. (2019). Epic: efficient private image classification (or: learning from the masters). In *Cryptographers' Track at the RSA Conference*, pages 473–492. Springer.

Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11):612–613.

Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708.

Turk, M. A. and Pentland, A. P. (1991). Face recognition using eigenfaces. In *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 586–591. IEEE.

Wang, M. and Deng, W. (2018). Deep face recognition: A survey. *arXiv preprint arXiv:1804.06655*.

Yao, A. C. (1982). Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE.

A Attachments

The source code for this project is available on the included USB and through GitHub.

There are two seperate smaller projects:

1. **Face matching**: This project includes the training, testing and design of the convolutional neural network used for face matching.
2. **Secure face matching**: This project includes the different secure operations as well as the MPC set up and the binaries for transforming regular Docker containers to MPC parties.

The first project can be found on the USB in the directory named *facematching* or via `https://github.com/Fluxmux/facematching`. The second project can be found on the USB in the directory named *securefacematching* or via `https://github.com/Fluxmux/securefacematching`.

FACULTY OF ENGINEERING TECHNOLOGY
DE NAYER (SINT-KATELIJNE-WAVER) CAMPUS
Jan De Nayerlaan 5
2860 SINT-KATELIJNE-WAVER, België
tel. + 32 16 30 10 30
fet.denayer@kuleuven.be
www.fet.kuleuven.be