

# NeuralHash for Privacy Preserving Image Analysis

BJØRN-INGE STØTVIG THORESEN

## SUPERVISORS

Dr. Vladimir Oleshchuk

Dr. Harsha Sandaruwan Gardiyawasam Pussewalage

**University of Agder, 2022**

Faculty of Engineering and Science

Department of ICT



# Abstract

This thesis aims to investigate how Apple’s NeuralHash algorithm can be used in the context of FR to improve privacy in FR systems. Existing FR solutions rely on having facial images available to match identities, however, this can impair the privacy of individuals, as the images can contain sensitive information that the individuals do not want to share. In this thesis, the NeuralHash algorithm is used to hash facial images of subjects in the ColorFERET Dataset, and the NeuralHashes are compared to attempt to identify the same subjects and different subjects. The NeuralHash algorithm’s ability to hide information is also investigated, in addition to collision- and evasion attacks on NeuralHash. The results show that using a threshold of approximately 0.24, the FAR and FRR are 9.68 %. If the threshold is set to 0.1, the FAR drops to 0.16 %, while the FRR rises to 31.45 %. Some general information about images such as gender can be inferred from the NeuralHash, while more nuanced information is not retrievable. Gradient-based attacks can be used against NeuralHash to evade collisions, and to force collisions with a target NeuralHash.



# Acknowledgments

I would like to thank my supervisors, Dr. Vladimir Oleshchuk and Dr. Harsha Sandaruwan for assisting me in the writing of this thesis. I received a lot of helpful input during discussions, which helped me progress in the writing process.

Portions of the research in this paper use the FERET database of facial images collected under the FERET program, sponsored by the DOD Counterdrug Technology Development Program Office [1, 2].



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Field of research . . . . .	2
1.3	Problem statement . . . . .	2
1.4	Research Objective . . . . .	2
1.5	Related work . . . . .	3
1.6	Contributions . . . . .	6
1.7	Outline . . . . .	6
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Perceptual Hashing . . . . .	7
2.2	NeuralHash . . . . .	8
<b>3</b>	<b>Solution</b>	<b>11</b>
3.1	Hypothetical scenario . . . . .	11
3.2	Solution architecture . . . . .	11
3.3	NeuralHash comparison . . . . .	12
3.4	Matching . . . . .	13

<b>4</b>	<b>Experiments</b>	<b>14</b>
4.1	Dataset . . . . .	14
4.2	Experimental setup . . . . .	16
4.2.1	Obtaining the NeuralHash model . . . . .	16
4.2.2	Cleaning the dataset . . . . .	17
4.2.3	Inference . . . . .	17
4.2.4	Hamming Distances . . . . .	18
4.3	Experiment 1 . . . . .	19
4.3.1	Implementation . . . . .	19
4.3.2	Results . . . . .	20
4.4	Experiment 2 . . . . .	26
4.4.1	Implementation . . . . .	26
4.4.2	Results . . . . .	28
4.5	Experiment 3 . . . . .	29
4.5.1	Implementation . . . . .	30
4.6	Results . . . . .	31
4.7	Experiment 4 . . . . .	32
4.7.1	Implementation . . . . .	32
4.7.2	Results . . . . .	36
4.8	Experiment 5 . . . . .	39
4.8.1	Results . . . . .	39
4.9	Experiment 6 . . . . .	40
4.9.1	Results . . . . .	41



<b>5</b>	<b>Discussion</b>	<b>44</b>
5.1	Experimental results summary . . . . .	45
5.2	Hypothetical scenario . . . . .	46
5.3	Future work . . . . .	48
<b>6</b>	<b>Conclusion</b>	<b>49</b>
	<b>References</b>	<b>53</b>
<b>A</b>	<b>Data cleaning</b>	<b>A-1</b>
<b>B</b>	<b>Inference</b>	<b>A-2</b>
<b>C</b>	<b>Calculate Hamming Distances</b>	<b>A-4</b>
<b>D</b>	<b>Average Hamming Distances and Standard Deviation</b>	<b>A-5</b>
<b>E</b>	<b>FAR &amp; FRR</b>	<b>A-8</b>
<b>F</b>	<b>Few_pixels Attack Script</b>	<b>A-11</b>

# List of Figures

- 2.1 Locality-sensitive hashing . . . . . 8
- 2.2 NeuralHash inference . . . . . 8
- 2.3 Contrastive loss . . . . . 9
  
- 3.1 Proposed solution . . . . . 12
- 3.2 HamD example . . . . . 12
  
- 4.1 Experiment 1 plot data list . . . . . 20
- 4.2 HamD distribution . . . . . 21
- 4.3 HamD distribution for subject 00106 . . . . . 22
- 4.4 HamD distribution for subject 00135 . . . . . 23
- 4.5 HamD distribution for subject 00407 . . . . . 24
- 4.6 HamD distribution for subject 00538 . . . . . 25
- 4.7 FAR and FRR plot . . . . . 31
- 4.8 Gender network structure . . . . . 34
- 4.9 Race network structure . . . . . 35
- 4.10 Gender model accuracy . . . . . 37
- 4.11 Gender model loss . . . . . 37
- 4.12 Race model accuracy . . . . . 38

4.13 Race model loss . . . . .	38
--------------------------------	----

# List of Equations

4.2	Average SelfDist and DiffDist. . . . .	27
4.4	Standard deviation ( $\sigma$ ), and variance ( $\sigma^2$ ). . . . .	27
4.5	False Acceptance Rate. . . . .	30
4.6	False Rejection Rate. . . . .	30

# List of Listings

- 4.1 Pseudocode for cleaning the dataset. The complete code is listed in Appendix A. . . . . 17
- 4.2 Pseudocode for performing inference with the NeuralHash model. The complete code is listed in Appendix B. . . . . 18
- 4.3 Pseudocode for calculating HamD. The complete code is listed in Appendix C. . . . . 18
- 4.4 Pseudocode for calculating average HamD and standard deviation. The complete code is listed in Appendix D . . . . . 27
- 4.5 First row of output from Experiment 2 . . . . . 28
- 4.6 Sample output from Experiment 2. The first four rows correspond to the subjects highlighted in Experiment 1. The remaining rows are sampled randomly from the 990 remaining subjects. . . . . 29
- 4.7 Pseudocode for calculating FAR and FRR. The complete code is listed in Appendix E . . . . . 30
- 4.8 Commands for generating images using the standard evasion attack. . . . . 41
- 4.9 Command for generating images using the edges\_only attack. . . . . 41
- 4.10 Command for generating images using the few\_pixels attack . . . . . 42
- A.1 Removing all non-frontal face images . . . . . A-1
- B.1 Code for the NeuralHash class used for inference. . . . . A-2

C.1	Method for calculating and storing Hamming Distances. . . . .	A-4
D.1	Method for calculating average HamD and standard deviation for each subject. . . . .	A-5
E.1	Method for aggregating the necessary data to plot average FAR and FRR.	A-8
E.2	Method for calculating error rates. . . . .	A-8
E.3	Method for counting true positives, false positives, true negatives and false negatives. . . . .	A-9
E.4	Method for calculating average FAR and FRR for current threshold. . . . .	A-10
E.5	Method for plotting average FAR and FRR for all threshold values. . . . .	A-10
F.1	Modified <i>adv2_few_pixels_attack.py</i> script. . . . .	A-11

# List of Tables

4.1	Subject poses in the ColorFERET Dataset. . . . .	15
4.2	Distribution of male and female subjects in the ColorFERET Dataset. . . . .	15
4.3	Distribution of race for the subjects in the ColorFERET Dataset. . . . .	15
4.4	Hyperparameters for the models. . . . .	35
4.5	Results from Adversary 1. . . . .	40
4.6	Results from Adversary 2. . . . .	42

# Abbreviations

**Adam** Adaptive Moment Estimation

**CCTV** Closed-Circuit Television

**CSAM** Child Sexual Abuse Material

**DFFN** Deep FeedForward Network

**DiffDist** Hamming Distance between NeuralHashes of images of different subjects

**EER** Equal Error Rate

**FAR** False Acceptance Rate

**FN** False Negative

**FP** False Positive

**FR** Facial Recognition

**FRR** False Rejection Rate

**GDPR** General Data Protection Regulation

**HamD** Hamming Distance

**IP** Internet Protocol



**LFW** Labeled Faces in the Wild

**LSH** Locality-Sensitive Hashing

**ML** Machine Learning

**MSE** Mean Squared Error

**PPML** Privacy-preserving Machine Learning

**PriMIA** Privacy-preserving Medical Image Analysis

**PSI** Private Set Intersection

**RQ** Research Question

**SelfDist** Hamming Distance between NeuralHashes of images of the same subject

**SSIM** Structural Similarity

**TN** True Negative

**TP** True Positive

**ZeroR** Zero Rate

# 1 | Introduction

Facial Recognition (FR) is an advanced technology used in many different applications such as advertising, healthcare and security. The technology is used to match digital images or video containing faces against a database of faces. This is achieved by first detecting faces in an image and distinguishing them from other objects, and then identifying the detected faces [3]. Research on FR goes as far back as 1963, when Woodrow Wilson Bledsoe attempted to apply pattern-recognition research to the problem of identifying a given face among many examples of faces [4]. A lot of research has been conducted in this area since then [5, 6, 7, 8], and many advances and breakthroughs have been achieved as a result. Furthermore, the rapid evolution of computer technology has dramatically increased the amount of data able to be processed. This has enabled data-driven paradigms such as Machine Learning (ML) to advance to the forefront of FR research.

## 1.1 Motivation

One drawback of traditional ML approaches for FR is that these systems typically require the images of faces to be available in order to function. FR systems can run locally on user devices, such as Apple's FaceID for unlocking an iPhone [9], or remotely on a server, such as images captured by IP CCTV camera systems, and sent to a remote server [10]. In cases where the FR system needs to match captured images against a database of facial images, either the database must be available on the device capturing images, or the captured images must be provided to the remote machine containing the database. This can be problematic if either the captured images or the database contain sensitive information which neither party is willing to share. One solution to this problem is to apply transformations to the data before sending it, such that it can still be used for FR, but the image data is concealed. The NeuralHash algorithm developed by Apple is a perceptual hashing algorithm used to create a fingerprint of an image, which hides the

image information while retaining enough information about the image to be useful for image analysis. The motivation for this thesis is to investigate whether NeuralHash can be used to protect image data while allowing for identification in FR systems.

## 1.2 Field of research

In order to investigate whether NeuralHash can be used to improve privacy in FR systems, research within the fields of FR technology, perceptual hashing, and Privacy-preserving Machine Learning (PPML) is essential. The research lies within the domain of FR, which means existing solutions for FR can be used as a benchmark. Since NeuralHash is a perceptual hashing algorithm, research on these types of systems is important in order to learn their advantages and disadvantages. This is useful for solving problems that may arise as a result of potential shortcomings of perceptual hashing systems. Finally, the research focuses on ML for privacy, hence research on PPML will be relevant for the thesis.

## 1.3 Problem statement

The main problem this thesis aims to solve is how to use NeuralHash to compare facial images without revealing the content of the images. Facial images fall under the category of biometric data according to the General Data Protection Regulation (GDPR) [11], and need to be protected accordingly. NeuralHash is a promising technique for image analysis, and it is worth investigating if it can be utilised to improve privacy in FR systems.

## 1.4 Research Objective

In order to solve the problem introduced in the Problem statement, it is necessary to investigate how NeuralHashes representing facial images can be compared, and used in classification. Furthermore, it is necessary to evaluate the classification performance of such a solution in order to get an idea of how well it works. It is also important to verify

that the NeuralHashes do not leak any information about the images they represent, other than the classification result. These research objectives lead to the formulation of the following Research Questions (RQs):

1. How can NeuralHash be used to identify whether two facial images belong to the same person?
2. How can the classification performance be evaluated?
3. Does NeuralHash reveal any information about the images other than the classification result?

## 1.5 Related work

The initial publication of the NeuralHash algorithm came as a part of a system for detecting Child Sexual Abuse Material (CSAM) in photos uploaded to an iCloud Photos account [12]. The system is designed to prevent Apple from learning anything about any of the uploaded benign photos, and only learn about the photos containing illegal CSAM content. The system uses a database of known CSAM images which has been transformed into an unreadable set of hashes using NeuralHash. This transformed database is stored on users' devices, and is used in a cryptographic protocol called Private Set Intersection (PSI). The protocol compares an image on the user device with the images in the transformed CSAM database, and generates a safety voucher which stores the match result. The safety voucher also encrypts the user image before being uploaded to iCloud Photos. Even if a match is positive, Apple will not be able to decrypt and view the content of the user image until a threshold of matches has been exceeded. This helps to prevent false positive matches from being viewed by Apple.

A case study of NeuralHash was conducted by Struppek et al. (2022) [13], where the security and privacy aspects of Apple's CSAM detection system was analysed. The paper focuses on how the NeuralHash algorithm can be exploited to produce false positive and

false negative matches. Since the all the NeuralHash model parameters and weights are publicly available, gradient-based adversarial attacks are possible to implement. These types of attacks are based on perturbing input images in such a way that the changes in the image are hardly visible to humans, but a neural network will classify the altered image differently than the original. The visual similarity between the input image and the perturbed image is measured using a Structural Similarity (SSIM) score ranging from 0 to 1, where a higher SSIM value denotes visually closer images. In the first experiment, a database of NeuralHashes is computed, and a target hash is selected according to the similarity between the NeuralHashes of the target image and an input image. The goal is to perturb the input image using the gradient of the NeuralHash model and the SSIM score, such that the NeuralHash matches the target hash, and the altered image is as close to the original as possible, visually. The next attack details how the gradient of the NeuralHash model can be used to evade detection of illegal content by perturbing an input image such that the NeuralHash differs from the original NeuralHash, up to a specified threshold of bits. This attack uses the same technique as the in the first experiment, and the visual representations of the original and altered images are negligible for humans when using a low threshold. The paper also demonstrates that NeuralHash retains some information about the image the NeuralHash was generated from. By creating a simple Deep FeedForward Network (DFFN) taking a NeuralHash as input, and predicting classes for objects depicted in the image the NeuralHash was generated from, the system is able to learn some of the inherent structure of NeuralHashes, and extract information about the type of object depicted in the image. The results show that the system is able to alter an input image to generate hash collisions such that the altered input image is visually hardly distinguishable from the original. The gradient-based evasion attack shows similar results, where the input image is visually indistinguishable from the altered image, and the NeuralHash bits differ up to a set threshold. Finally, the results from the hash information extraction experiment show that out of 1,000 classes, an image class was correctly classified in 4.34 % of cases, where random guessing would have resulted in an accuracy of 0.1 % [13].

Microsoft has designed its own image-identification technology called *PhotoDNA* [14]. Similar to Apple’s NeuralHash, the goal of PhotoDNA is to combat the spread of CSAM by creating signatures of images, and using these to match against known CSAM images. The signature is a hash which is created by first converting the image to grayscale, resizing it, and breaking it into a grid. For each grid cell, a histogram of intensity gradients (edges) is calculated, and this histogram data replaces the pixels in the grid cell. The new image constitutes the image’s PhotoDNA [15].

In 2015, Google published a paper called *FaceNet: A Unified Embedding for Face Recognition and Clustering* [6]. The aim of this paper was to shift the trend of treating FR problems as classification problems, and instead treat them as a clustering problem. This is achieved by producing an embedding of a face, meaning a dense vector representation of the face. This embedding is projected to the unit hypersphere [16], and from here, the  $L_2$  distance to other face embeddings can be measured. Similar faces will have a low  $L_2$  distance, while different faces will have a higher  $L_2$  distance. The neural network used for the embeddings use triplet loss, which teaches the network to embed similar images close together, and different images further apart. This is done by using a triplet of images, where image one is an anchor image (or ground truth), image two is a positive image, meaning a different image of the same person, and image three is a negative image, which is an image of a different person. The distance between the anchor and the positive should be less than the distance from the anchor to the negative, by a set threshold. When testing the performance of the network on the Labeled Faces in the Wild (LFW) dataset, the network achieved a classification accuracy of  $99.63\% \pm 0.09$ .

A lot of research has been conducted within the field of ML in the context of FR, however, the majority of these works focus on the performance of the ML algorithms as opposed to their impact on privacy. One work that addresses privacy related to image analysis is the work of Ziller et al. (2020) [17]. In the paper, a software framework called Privacy-preserving Medical Image Analysis (PriMIA) is introduced and evaluated. The framework builds on PPML, and uses inference-as-a-service and end-to-end encryption to

provide a solution for keeping patient data private while simultaneously not exposing the model performing inference. The results show that the federated secure model is able to outperform human experts while learning nothing about the input data for the inference. This is possible due to a "Function Secret Sharing protocol expanded and adapted for neural networks".

## 1.6 Contributions

This work aims to contribute to the field of FR systems with focus on privacy. NeuralHash can be a good alternative solution to existing FR solutions, as image data can remain hidden during analysis. This work shows how the NeuralHash algorithm can be used for identity matching, how the performance of this solution can be evaluated, and how well NeuralHash is able to hide information about gender and race of individuals. This thesis lays the groundwork for further research on privacy preserving image analysis using NeuralHash, and deep perceptual hashing algorithms in general.

## 1.7 Outline

In Chapter 2, this thesis will address the background of the perceptual hashing and NeuralHash technologies. Thereafter, in Chapter 3 the architecture of the proposed solution will be presented. Next, in Chapter 4 the thesis presents the experiments used to evaluate the Research Questions presented in 1.4. This chapter details the implementation and the results of the experiments. Following this is Chapter 5, which will discuss the results obtained from the experiments, and the thesis in general. Finally, in Chapter 6 the conclusions are presented.

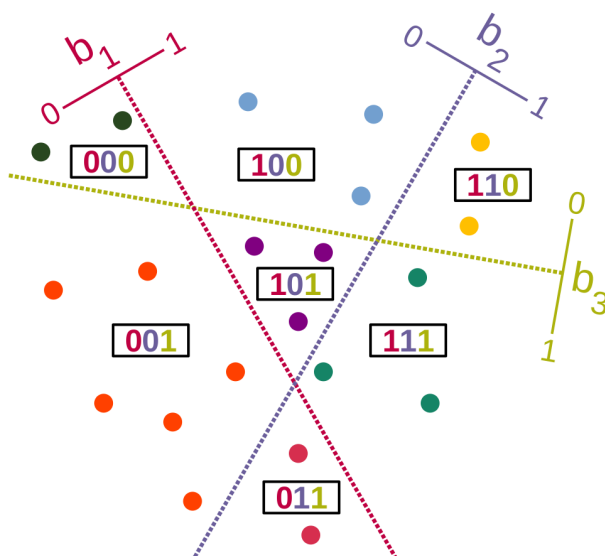
## 2 | Background

This chapter aims to explain the background on the technologies used for the research in this thesis. First, perceptual hashing will be explained, followed by NeuralHash, which is a deep perceptual hashing algorithm.

### 2.1 Perceptual Hashing

Perceptual hashing is a type of hashing algorithm which is used to generate fingerprints of various forms of multimedia [18]. Perceptual hashing is a type of Locality-Sensitive Hashing (LSH), which means similar inputs have similar or identical output. This is done by mapping inputs into different "buckets", such that each bucket encapsulates similar features in different inputs. The number of buckets in LSH is significantly smaller than the number of possible inputs, and similar inputs are mapped into the same bucket. This allows for data clustering and nearest neighbor search. The distance between inputs can also more easily be modelled, as polar inputs will be mapped to buckets that are far apart, meaning the relative distance between inputs and outputs are preserved [19]. This technique is useful to reduce the dimensionality of data, where the input data is too high-dimensional to be compared directly. Figure 2.1 illustrates the concept of mapping input data into buckets. The three hyperplanes  $b_1, b_2$  and  $b_3$  split the input space into distinct buckets, where hyperplane  $b_n$  is dedicated to bit  $n$  in the output. Each hyperplane has a positive and negative side, and the output bit corresponding to the hyperplane will be set to 0 if the input data is located on the negative side of the hyperplane, and 1 if it is on the positive side.

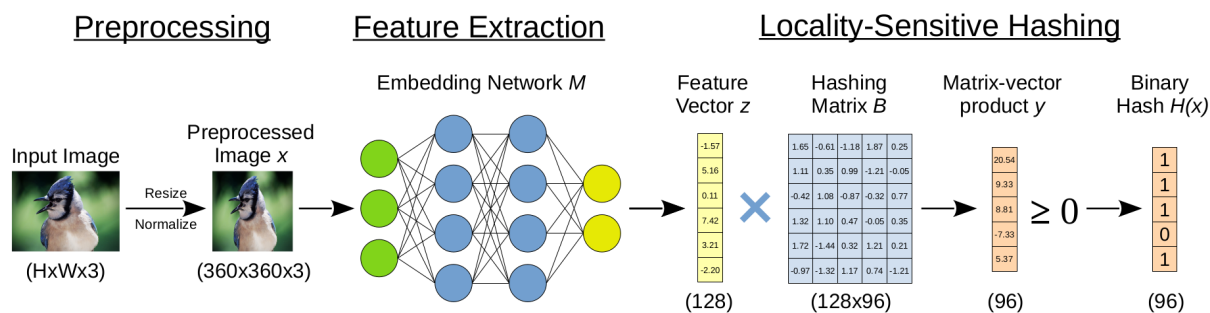




**Figure 2.1:** Locality-Sensitive Hashing mapping input data into different buckets [13].

## 2.2 NeuralHash

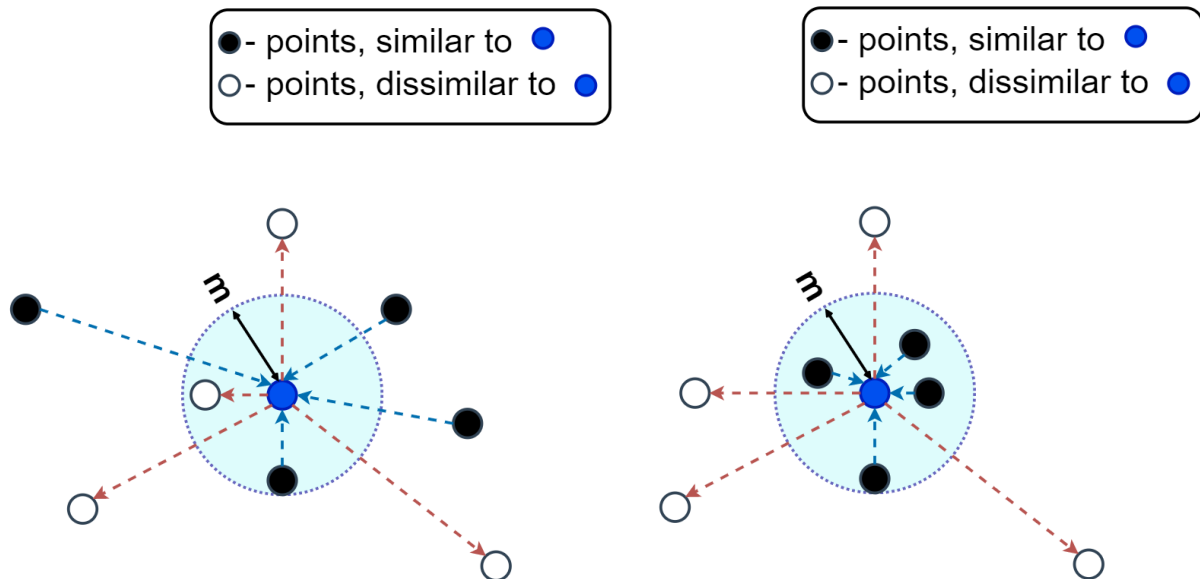
NeuralHash is a deep perceptual hashing algorithm, meaning it utilizes a deep neural network to generate hash values. Figure 2.2 shows an overview of the steps performed in the calculation of NeuralHashes.



**Figure 2.2:** NeuralHash inference. The image is preprocessed before being fed to an embedding network. The network maps the input image to an abstract feature representation. The LSH step uses a Hashing Matrix which defines 96 hyperplanes to map the feature vector into one of two buckets for each hyperplane. If a vector product is positive, the corresponding output bit is mapped to 1, if it is negative, it is mapped to 0 [13].

NeuralHash takes RGB images of shape  $360 \times 360 \times 3$  as input, with pixel values in the range  $[-1, 1]$ . The embedding network is based on a modified MobileNetV3 architecture, and has 1.8 million parameters [13]. The network has been trained in a self-supervised learning scheme [13], meaning it has seen some labeled examples, and used the learned structure of this data to label and train on unlabeled examples [20].

The loss function is contrastive loss, which is a way to train the network to cluster data. This is achieved by creating positive and negative pairs of samples for each training image, where the training image acts as the anchor image. A positive pair consists of the anchor image, and a transformed version of the training image, which ensures the pair is similar. A negative pair consists of visually close images that are from different classes, meaning the network should learn to distinguish them from one another. The network is trained to generate close feature vectors for positive pairs, and distant vectors for negative pairs [13]. Figure 2.3 illustrates the learning problem for contrastive loss.



**Figure 2.3:** Contrastive loss. Similar datapoints outside the margin  $m$  are moved closer to the anchor, while dissimilar datapoints within the margin are moved further away from the anchor [21].

Perceptual hashing algorithms usually consist of two steps [13]:

1. Extract visual features from the input
2. Encode features into a feature vector

The feature vector is an abstract representation of the input image, and consists of vectors of numbers that encode the features of the image.

The next step is LSH, where feature vectors are mapped to buckets such that close feature vectors end up in the same bucket. This can be achieved by defining a (random) hyperplane for each bit  $i$  in the output, and checking which side of the  $i$ -th hyperplane the corresponding feature vector lies. In Figure 2.2, the hyperplanes are defined in the Hashing Matrix, and for each feature vector, the dot product of the vector and the hashing matrix is calculated to produce a matrix-vector product. The matrix-vector product is a vector of real numbers, and a Heavside step function is applied to each element in the vector to produce the NeuralHash [13]. Figure 2.1 is a 2D representation of mapping input feature vectors into different buckets, although in the NeuralHash algorithm, this LSH mapping is 96-dimensional.

## 3 | Solution

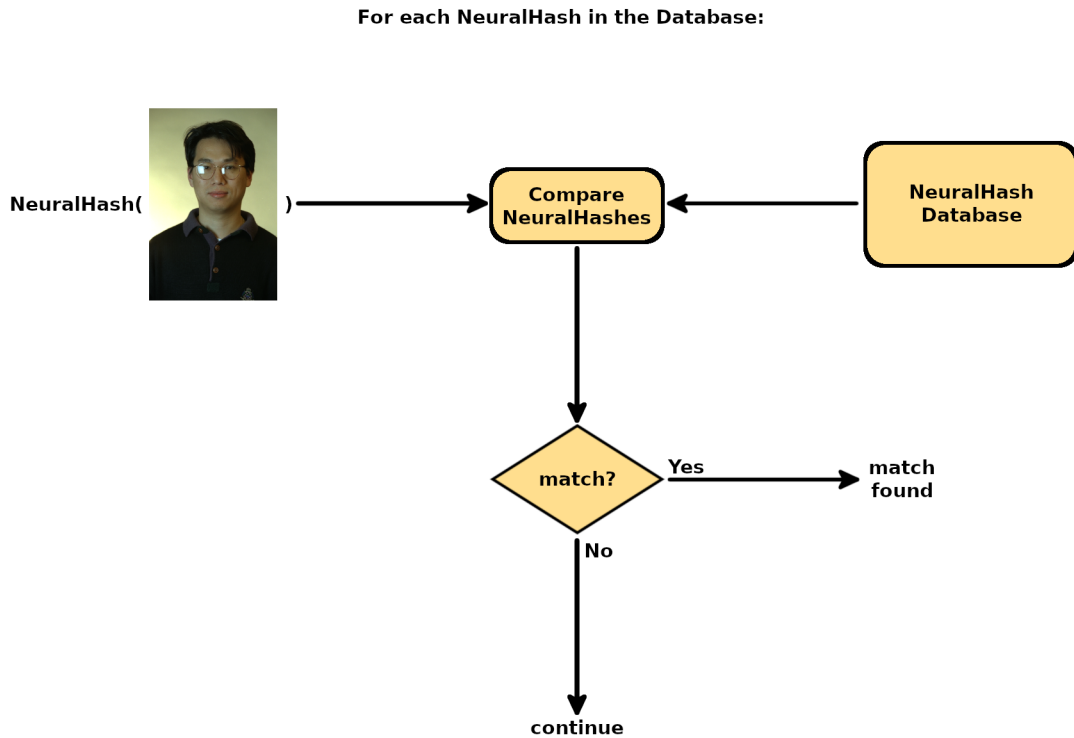
This chapter will introduce a hypothetical scenario in order to present the proposed solution for the problem. This is achieved by showcasing how NeuralHashes can be compared, and how the classification performance can be evaluated.

### 3.1 Hypothetical scenario

A hypothetical scenario is introduced to elaborate on a possible use case for a system making use of the NeuralHash algorithm for identity matching of facial images. The setting of this scenario is airport security, and the system using NeuralHash will be deployed to identify known persons of interest. The system has access to a database containing the NeuralHashes of facial images of the known persons of interest, and uses this database for image comparisons. Facial images of people boarding planes are taken, and the NeuralHash of each image is calculated. The NeuralHashes of the captured images are compared to the NeuralHashes stored in the database of persons of interest, and if the similarity of the NeuralHashes are within a set threshold, the person is flagged, which leads to additional security measures for the individual.

### 3.2 Solution architecture

The architecture of the solution describes how a system using NeuralHash to perform identity matching can be structured. Figure 3.1 shows the general architecture of the solution. The NeuralHash Database contains NeuralHashes of persons of interest which the system should detect. If a match is found, the person being examined is considered to be present in the database of persons of interest.



**Figure 3.1:** Diagram of the proposed solution.

### 3.3 NeuralHash comparison

The solution consists of evaluating and comparing NeuralHashes of various images of faces. Each NeuralHash pair is compared by calculating the Hamming Distance (HamD) between the 96-bit NeuralHashes. The HamD between two strings of the same length is the number of positions the corresponding character is different [22]. Thus, the HamD is a measure of how similar two strings are, as seen in Figure 3.2.

```

00098_931230_fa.ppm 0010000001000101101000010110100001110100100011101101100101001001010100000000111110000100011110
00098_931230_fb.ppm 0010000000000101101000010110100001110100100011101101100101001001010100000000111110000100011110
dist                3
  
```

**Figure 3.2:** Example of the HamD between the NeuralHashes of two images.

## 3.4 Matching

If two distinct images have similar NeuralHashes, the images have a lot of features in common. If one image contains the face of an individual, and the NeuralHash is almost identical to the NeuralHash of another image, the images most likely contain the face of the same person. However, due to variety in facial expression, hairstyle, accessories such as glasses etc., it is best to assume there will be some discrepancy between NeuralHashes of different images of the same person. Hence, a *threshold* is introduced to determine how similar two NeuralHashes must be in order to facilitate a successful match. Since the threshold is a measure of similarity between NeuralHashes, it can use Hamming Distance as its metric.

In order to evaluate the classification performance of the system, the threshold can be varied, and for each threshold value, the False Acceptance Rate (FAR) and False Rejection Rate (FRR) can be recorded. This will give insight into which threshold results in the best classification performance.

# 4 | Experiments

This chapter aims to elaborate on the experiments implemented for this thesis. The dataset used for the experiments and the experimental setup is described first, followed by the implementation and results from each experiment.

The experiments are used to investigate the RQs defined in Section 1.4. Four experiments are conducted in order to investigate the three RQs, along with two additional experiments. Experiment 1 investigates RQ 1, experiments 2 and 3 investigate RQ 2, and experiment 4 investigates RQ 3. The outcome of these experiments will be used as a basis for evaluating the RQs. The two additional experiments are conducted in order to discuss the overall security and privacy aspects of the NeuralHash algorithm.

The python implementation of the experiments is available on GitHub at <https://github.com/bjotho/AppleNeuralHash>.

## 4.1 Dataset

The dataset used for the experiments is the ColorFERET Dataset [23]. The dataset contains facial images of 994 subjects, from different angles and with a variety of facial expressions. The dataset is compressed into a tar file, and all the images for a given subject are located in a folder named after the five-digit ID corresponding to the subject. Each image file is compressed with bzip2 [24] to create bz2 files, and the image format is ppm.

The naming scheme of the image files is as follows: *SubjectID\_session\_pose*. There are a total of 13 different poses, which are listed in Table 4.1.

fa	Regular frontal image
fb	Alternative frontal image, taken shortly after the corresponding fa image
pl	Profile left
hl	Half left - head turned about 67.5 degrees left
ql	Quarter left - head turned about 22.5 degrees left
pr	Profile right
hr	Half right - head turned about 67.5 degrees right
qr	Quarter right - head turned about 22.5 degrees right
ra	Random image - head turned about 45 degrees left
rb	Random image - head turned about 15 degree left
rc	Random image - head turned about 15 degree right
rd	Random image - head turned about 45 degree right
re	Random image - head turned about 75 degree right

**Table 4.1:** Subject poses in the ColorFERET Dataset.

The dataset also comes with labels for each image, which denote the gender and race of the depicted subject. Tables 4.2 and 4.3 show the distribution of gender and race in the dataset respectively. The No. subjects column denotes how many subjects belong to a particular category, while the No. images column denotes how many images from the given categories in the dataset were used for the experiments.

	No. subjects	No. images
Male	591	1715
Female	403	1007

**Table 4.2:** Distribution of male and female subjects in the ColorFERET Dataset.

	No. subjects	No. images
Asian	171	485
Asian-Middle-Eastern	53	151
Asian-Southern	1	2
Black-or-African-American	78	215
Hispanic	57	138
White	618	1691
Native-American	2	6
Pacific-Islander	10	26
Other	4	8

**Table 4.3:** Distribution of race for the subjects in the ColorFERET Dataset.



## 4.2 Experimental setup

The experimental setup describes the prerequisites for performing the experiments. This entails how to obtain the NeuralHash model, cleaning the dataset, performing inference with the NeuralHash model, and calculating Hamming Distances.

### 4.2.1 Obtaining the NeuralHash model

In order to calculate the NeuralHash of an image, the NeuralHash model is needed. For this thesis, the model was implemented in Python by following the tutorial by Asuhariet Ygvar [25]. First, the Python dependencies are installed. Next, the NeuralHash model is obtained by downloading the IPSW file for iOS 15.2.1 (19C63) for iPhone 13 [26]. This file contains the firmware of iOS 15.2.1 for iPhone 13, which includes the NeuralHash model. After unpacking the file, the largest .dmg file is mounted, and the NeuralHash model files can be copied from the path `/System/Library/Frameworks/Vision.framework/` in the mounted system volume. The four required files are:

- `neuralhash_128x96_seed1.dat`
- `NeuralHashv3b-current.espresso.net`
- `NeuralHashv3b-current.espresso.shape`
- `NeuralHashv3b-current.espresso.weights`

The files are compressed with the LZSFE compression algorithm, which is why the next step is to install LZSFE and decompress the files. A C implementation of LZSFE can be installed from [27]. Finally, the model can be converted to ONNX by using a fork of the deep learning inference framework TNN [28]. This results in a `model.onnx` file which can be used together with the `neuralhash_128x96_seed1.dat` file for inference. Running the

nnhash.py script provided by Ygvar with a sample image yields a hexadecimal NeuralHash of the image as output.

## 4.2.2 Cleaning the dataset

The ColorFERET Dataset contains facial images with many different poses. For this experiment, the only images used are the frontal images, which specifically means the image files containing "f" in the *pose* part of the filename. The dataset is therefore cleaned accordingly, see Listing 4.1.

**Listing 4.1:** Pseudocode for cleaning the dataset. The complete code is listed in Appendix A.

```
1 for image in dataset:
2     if 'f' is not in image filename:
3         delete image
```

## 4.2.3 Inference

In order to allow inference of multiple images, the relevant code was moved to a NeuralHash class containing the methods *calculate\_neuralhash* and *im2array*, both taking an image path as argument.

The first step for calculating the NeuralHash is to convert the input image to a Numpy array, which happens in the *im2array* method. This method starts by converting the image to RGB. Next, the image is resized to  $360 \times 360$ . Finally, the RGB values are normalized to the range  $[-1, 1]$  [25].

The next step is to perform inference on the NeuralHash model. The output is a vector of 128 float values. The dot product of this output vector and the  $96 \times 128$  hashing matrix retrieved from the *neuralhash\_128x96\_seed1.dat* file is calculated. A binary step function

is applied to the resulting vector of 96 float values, where positive values (including 0) are converted to a binary 1 and negative values are converted to a binary 0. The vector of 1's and 0's is converted to a string of 96 bits for the binary representation, and a string of 24 hexadecimal values for the hexadecimal representation [25]. In the experiments, the resulting hashes are stored in a dictionary called *hash\_dict*. Listing 4.2 shows how the inference process is implemented.

**Listing 4.2:** Pseudocode for performing inference with the NeuralHash model. The complete code is listed in Appendix B.

```

1 def calculate_neuralhash(image_path):
2     calculate im2array(image_path)
3     calculate NeuralHash
4     calculate dot product of output array and 96x128 matrix from seed1
5     calculate binary step to get binary NeuralHash
6     format to hex for hexadecimal representation
7     return hex_hash, bin_hash
8
9 def im2array(image_path):
10    Open image from image_path and convert to RGB
11    Resize image to 360x360
12    Normalize RGB values to range [-1, 1]
13    return normalized values

```

#### 4.2.4 Hamming Distances

The metric used for measuring image similarities is the Hamming Distance between the NeuralHashes. Each image is compared to every other image, and the HamD is stored in a dictionary where the keys are the image filenames, and the values are dictionaries containing distances to every other image. The Levenshtein C extension module is used to calculate Hamming Distances [29].

**Listing 4.3:** Pseudocode for calculating HamD. The complete code is listed in Appendix C.

```

1 create dictionary for storing hamming distances
2 for hash1 in hash_dict:

```

```
3     create slice of hash_dict containing remaining hashes
4     for hash2 in remaining_hashes:
5         calculate hamming distance between hash1 and hash2
6         store distance in hamming_distances
```

## 4.3 Experiment 1

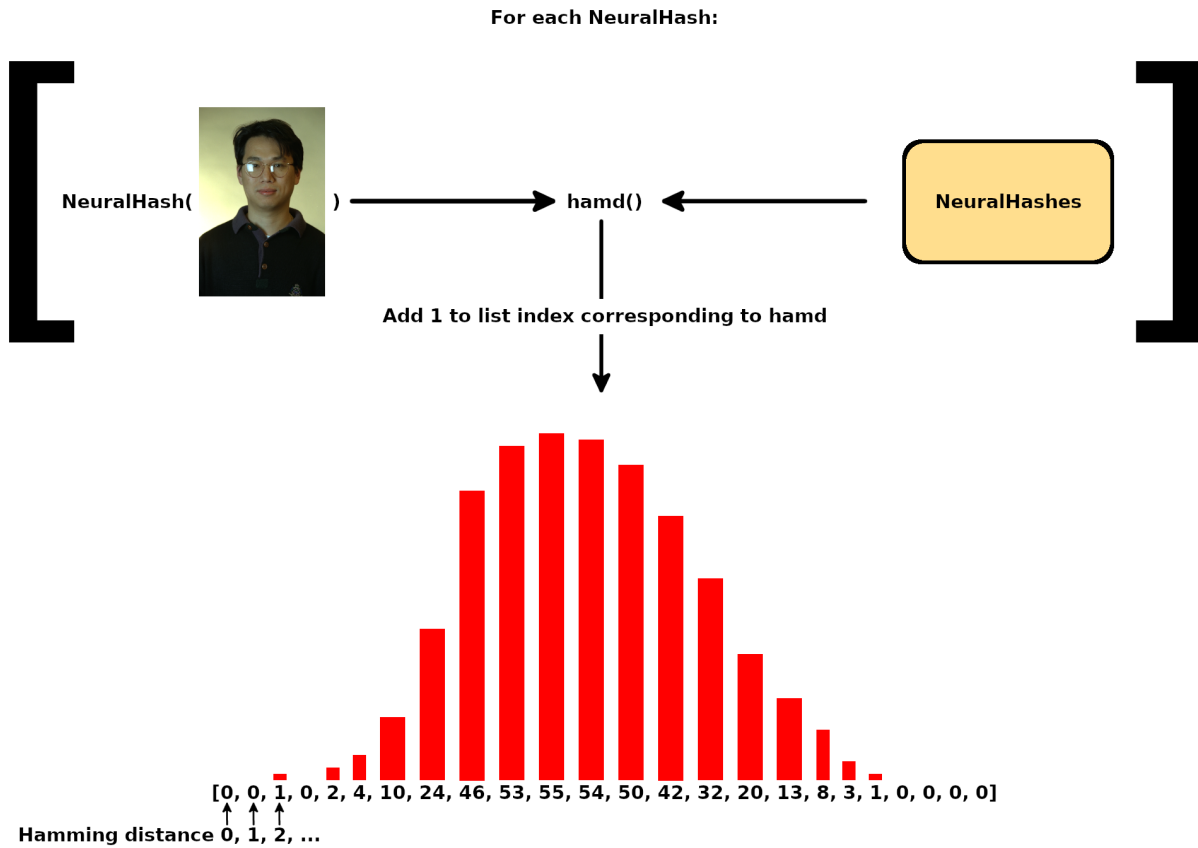
The first experiment examines the relationship between the HamD from one image to all other images, and the proportion of images with a given distance. In this experiment, images of the same subject are distinguished from images of different subjects, such that the Hamming Distance between NeuralHashes of images of the same subject (SelfDist) and Hamming Distance between NeuralHashes of images of different subjects (DiffDist) can be more easily visualized. The goal of this experiment is to investigate whether images of the same subject are reliably clustered close together, meaning they will have a low HamD compared to images of different subjects.

### 4.3.1 Implementation

In order to examine the relationship between HamDs of NeuralHashes, and the proportion of images with a given HamD, a plot is created for each subject to visualize this relationship. Each plot contains two subplots, where the blue plot represents the SelfDists for the current subject, and the red plot represents DiffDists for the current subject. One additional plot is created which aggregates all comparisons into one graph.

The plot data is created by iterating over the dictionary of Hamming Distances, and creating two lists of zeroes for each subject. The first list is used to count SelfDists, while the second list is used to count DiffDists. The length of the lists are equal to the length of the NeuralHashes plus one, because each element's index represents the HamD, and the element itself counts the number of images with the corresponding HamD. After inserting every image comparison into the lists, each list is normalized to the range  $[0, 1]$ , and the

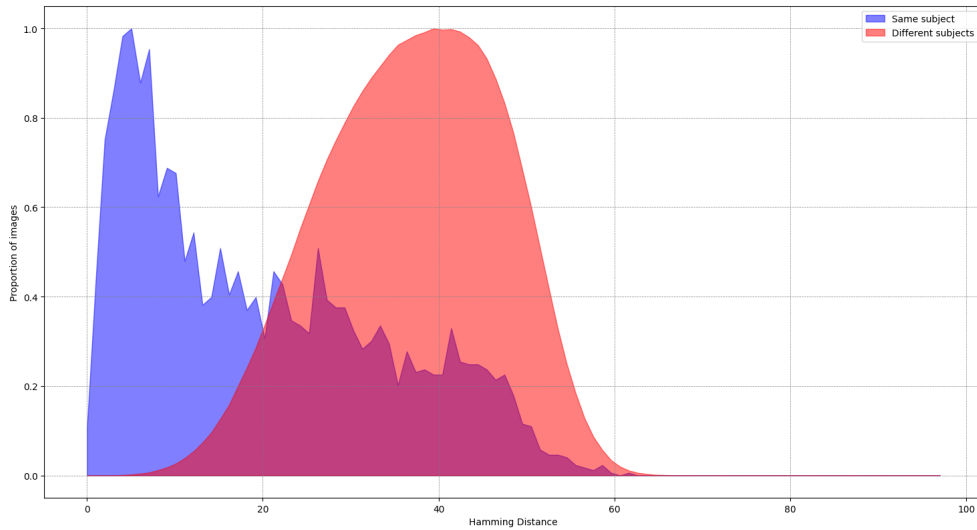
lists are used to plot the graphs. Figure 4.1 illustrates how this process works.



**Figure 4.1:** Creation of plot data list for one subject.

### 4.3.2 Results

The results from Experiment 1 show that the NeuralHash algorithm is able to cluster images of the same subject closer together than images of different subjects. Figure 4.2 shows how many comparisons between images of the same subject and images of different subjects have a given HamD proportionally.

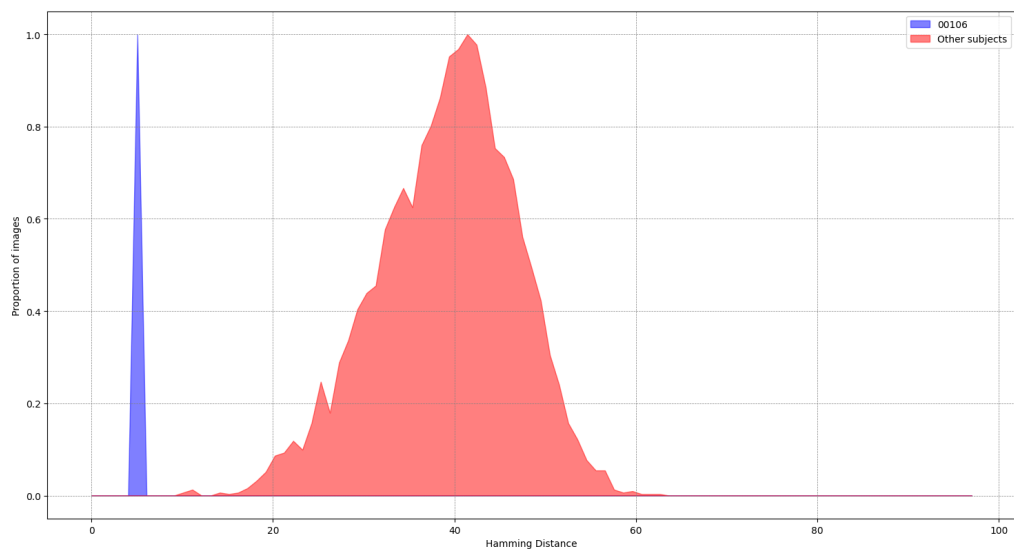
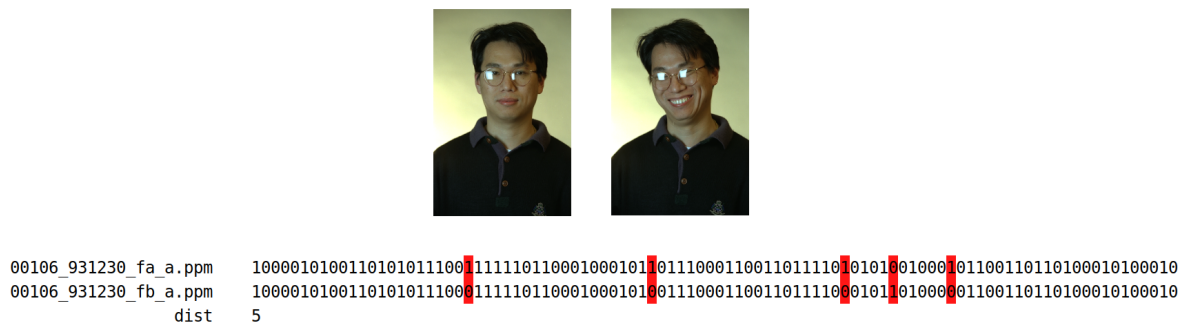


**Figure 4.2:** Distribution of HamDs. The x-axis represents the HamD, where  $x=0$  means a HamD of 0, while the y-axis shows how large proportion of the comparisons resulted in a given HamD, where 0 is the least and 1 is the most.

Figure 4.2 shows that images of the same subject (blue) in general have a lower HamD than images of different subjects (red). The graph shows that the most common SelfDist is around 5, while the most common DiffDist is approximately 40. The SelfDists are more spread out than the DiffDists, and the SelfDist graph is more jagged. This could be a result of the relative quantity of comparisons, as 743 of the 994 subjects only have two images, and thus only one comparison to themselves. In contrast, the number of comparisons to other subjects for each subject is  $(994 - N_s) \times N_s$ , where  $N_s$  is the number of images for subject  $s$ .

Figures 4.3, 4.4, 4.5 and 4.6 show examples of comparisons between images of subjects 00106, 00135, 00407 and 00538 respectively. Figures 4.3 and 4.5 show that the NeuralHash algorithm is able to extract many similar features from a facial image, even with differing facial expressions and variety in the hairstyle. In Figure 4.4, the two images of subject 00135 have different lighting, which is likely why the HamD is over 20. Still, the HamD between these two images are among the lowest of all the comparisons for this subject,

which means means the algorithm recognizes that these two images are more similar than the majority of the remaining images. Figure 4.6 shows that the NeuralHash algorithm can recognize similar faces even when subjects switch between wearing glasses and not.



**Figure 4.3:** HamD between images of subject 00106 (blue), and others (red).



```
00135_931230_fa_a.ppm 101101110010011100001010101011011111100001010011001000100100101001100110011001100000110010000101010
00135_931230_fb_a.ppm 1010001110110111000010101101111111000001001101100110011001000010101000000001010
dist 21
```

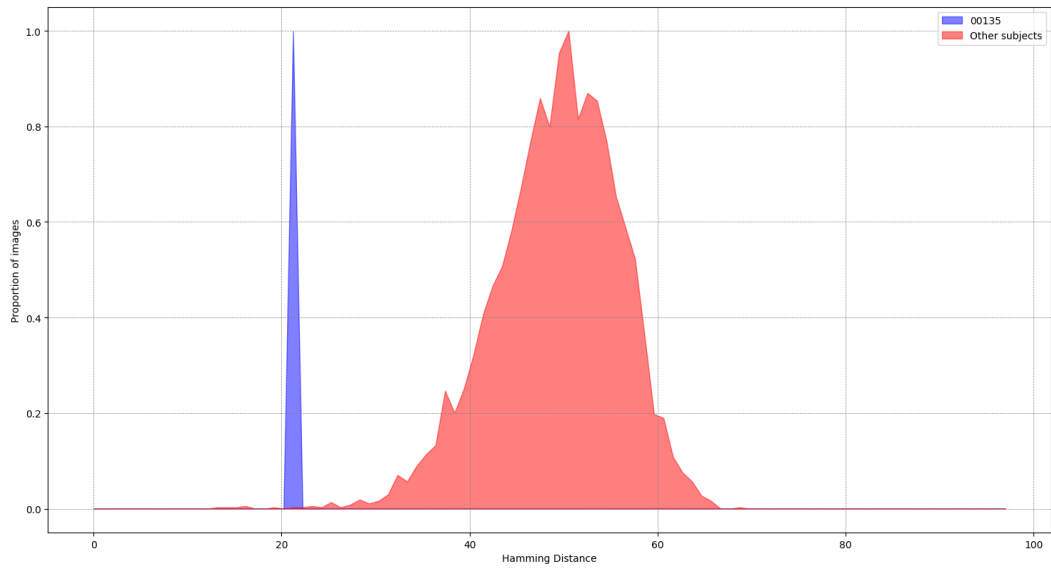
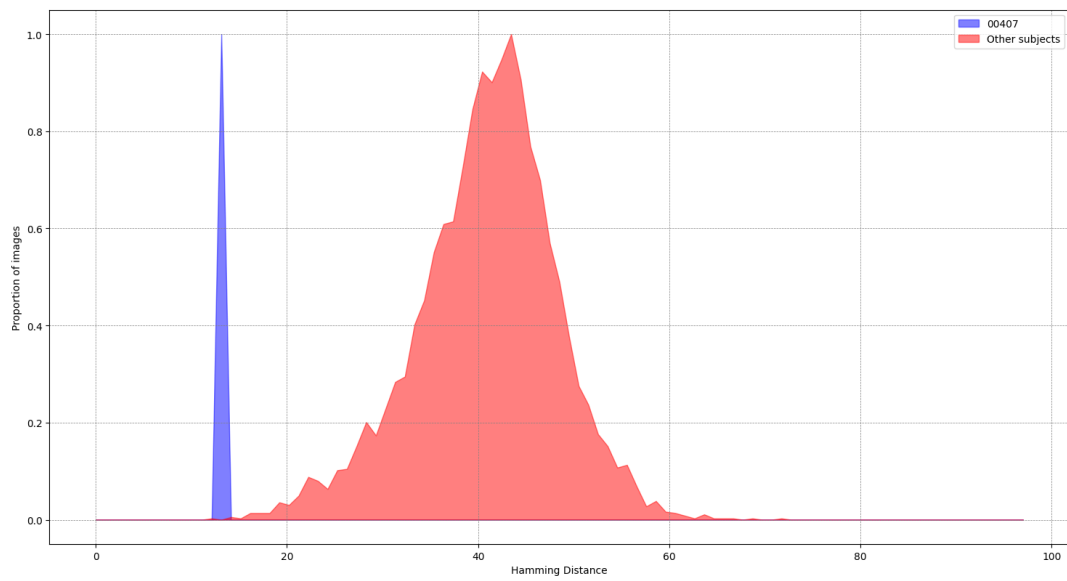


Figure 4.4: HamD between images of subject 00135 (blue), and others (red).

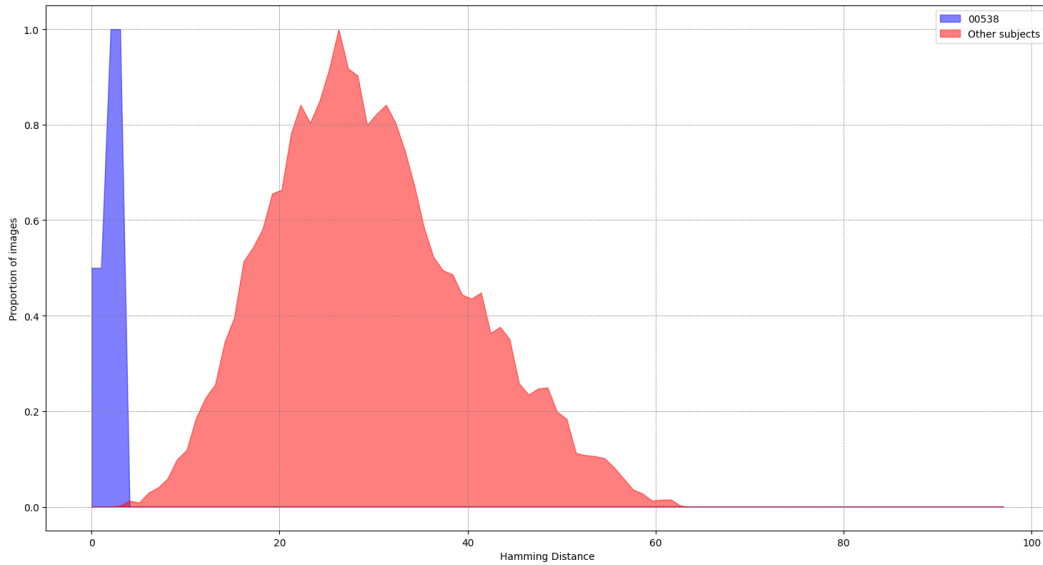
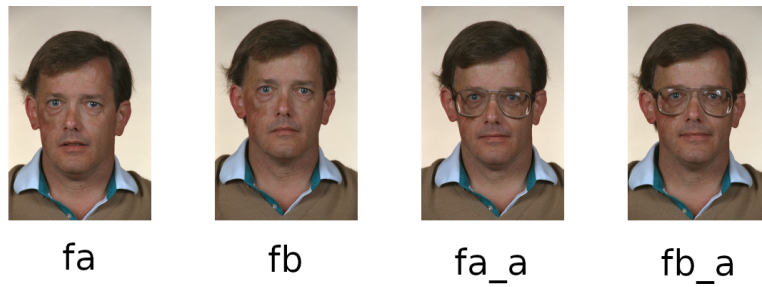




```
00407_940422_fa.ppm 010000000110100010110001000010001101110111010011010111010110000100101101110111000001010100010
00407_940422_fb.ppm 01000000001010001111000101000001011101101110100110101110101100001010100000
dist 13
```



**Figure 4.5:** HamD between images of subject 00407 (blue), and others (red).



```

00538_940519_fa.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
00538_940519_fb.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
dist 0

00538_940519_fa_a.ppm 10001001001000011011001101111000011100111010111111111111100100010001001001101011111100001010001011
00538_940519_fb_a.ppm 10001001001000011011001101111000011100111010111111111111100100010001001001101011111100001010001011
dist 1

00538_940519_fa.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
00538_940519_fa_a.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
dist 2

00538_940519_fa.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
00538_940519_fb.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
dist 2

00538_940519_fa.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
00538_940519_fb_a.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
dist 3

00538_940519_fb.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
00538_940519_fb_a.ppm 10001001001000011011001101111000011100111011111111111111100100010001001001101011111100001010001010
dist 3
    
```

Figure 4.6: HamD between images of subject 00538 (blue), and others (red).

## 4.4 Experiment 2

The second experiment adds to the first experiment by calculating the average SelfDist and DiffDist for each subject, and for all subjects. Additionally, the standard deviation is calculated for comparisons between the same subject and different subjects. This experiment will show which subjects' images the algorithm clusters close together, and which subject's images the algorithm considers far apart, as well as how widely spread the Hamming Distances are.

### 4.4.1 Implementation

For Experiment 2, a dictionary for storing average distances is created, and it counts four values for each subject. The values are:

1. Sum of HamD between images of the same subject
2. Sum of HamD between images of current subject and other subjects
3. Total number of comparisons between images of the same subject
4. Total number of comparisons between images of the current subject and other subjects

The average is calculated according to 4.1 and 4.2.  $x_i$  denotes the HamD between the current image and image  $i$  of the same subject, while  $x'_i$  denotes the HamD between the current image and image  $i$  of another subject.

$$avg_{self} = \frac{\sum x_i}{Total_{self}} \quad (4.1)$$

$$avg_{diff} = \frac{\sum x'_i}{Total_{diff}} \quad (4.2)$$

The standard deviation is calculated according to 4.3, where  $x$  denotes a datapoint value,  $\mu$  denotes the average value of the collection, and  $N$  denotes the size of the collection. First a list is created containing the numerator of the fraction in 4.3 without calculating the sum, hence  $(x_i - \mu)^2$ . The list contains lists of two elements each, one for images of the same subject, and one for images of different subjects. Next, the variance is calculated for same and different subjects, and since the variance is the standard deviation squared, the standard deviations for the same and different subjects are calculated by taking the square root of both the variances. Listing 4.4 shows the implementation of calculating the average HamD and standard deviation.

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}} \quad (4.3)$$

$$\sigma^2 = \frac{\sum(x_i - \mu)^2}{N} \quad (4.4)$$

**Listing 4.4:** Pseudocode for calculating average HamD and standard deviation. The complete code is listed in Appendix D

```

1 create empty dictionaries to store average hamd and standard deviation
2 for distance in stored hamming distances:
3     calculate sum of hamd between images of same subject
4     calculate sum of hamd between images of current subject and other subjects
5     count total comparisons between same subject
6     count total comparisons between images of current subject and other subjects
7     append values to average hamd dictionary
8
9 for tally in average hamd dictionary:
10    calculate sum_same / total_same

```

```

11     calculate sum_different / total_different
12     store temporarily in standard deviation dictionary
13
14 create new dictionary to store numerators for standard deviation formula
15 for distance in stored hamming distances:
16     calculate and append numerator using average hamds from standard deviation dictionary
17
18 for numerator in numerator dictionary:
19     calculate variance for comparisons between same subject
20     calculate variance for comparisons between current subject and other subjects
21     calculate standard deviation for comparisons between same subject
22     calculate standard deviation for comparisons between current subject and other
    ↪ subjects
23
24 for subject in standard deviation dictionary:
25     output average hamd and standard deviation for subject

```

## 4.4.2 Results

Listing 4.5 shows the first row of output from running Experiment 2. The output shows the average SelfDist and DiffDist, in addition to the standard deviation for comparisons between images for the same subject and different subjects. The results from Experiment 2 nicely reflect the results from Experiment 1, in the sense that the average SelfDist is lower than the average DiffDist. The listing also reflects that the SelfDists are more spread out than the DiffDists.

**Listing 4.5:** First row of output from Experiment 2

	Avg dist self	$\sigma$ self	Avg dist others	$\sigma$ others
1 All:	19.38	14.25	36.61	9.90

Listing 4.6 shows the output from Experiment 2 related to the subjects highlighted in Experiment 1, in addition to 10 randomly sampled subjects. The output is sorted on the average SelfDist, from low to high. In all of the samples, the average SelfDist is lower than the average DiffDist. This again indicates that the NeuralHash algorithm is able to

cluster images of the same subject more closely together than images of different subjects. The standard deviation for the same subject is 0 in most cases. This is because most of the subjects in the dataset only have two images, and therefore only one comparison. For the subjects with more than two images, the standard deviation can become quite large, such as for subjects 00596 and 00013. The reason for this could be images taken in different photo sessions with different lighting and the subject wearing different clothes.

**Listing 4.6:** Sample output from Experiment 2. The first four rows correspond to the subjects highlighted in Experiment 1. The remaining rows are sampled randomly from the 990 remaining subjects.

	Avg dist self	$\sigma$ self	Avg dist others	$\sigma$ others
1				
2	00538: 1.83	1.07	29.27	10.33
3	00106: 5.00	0.00	38.66	7.51
4	00407: 13.00	0.00	40.42	7.21
5	00135: 21.00	0.00	48.70	6.57
6				
7	00684: 1.00	0.00	36.10	8.13
8	00467: 5.00	0.00	32.43	9.85
9	00305: 7.00	0.00	36.03	10.42
10	01001: 7.00	0.00	41.50	6.28
11	00944: 7.00	0.00	42.13	9.31
12	00889: 12.00	0.00	38.07	7.71
13	00910: 16.00	0.00	34.87	7.70
14	00854: 16.00	0.00	46.62	5.40
15	00596: 27.00	17.02	33.60	10.29
16	00013: 35.50	20.55	41.47	9.19

## 4.5 Experiment 3

In the third experiment, all the collected data is utilized to calculate the False Acceptance Rate and False Rejection Rate. A HamD threshold is used to determine if a given pair of images are considered the same subject, and the threshold is varied to find the best trade-off between accuracy and leniency.

### 4.5.1 Implementation

For each subject, one image is selected to best represent that subject in order to minimize the FAR. Each image's NeuralHash is compared with all the other NeuralHashes of the other images. The images are labeled with **True** or **False** to indicate if they belong to the same subject. A *threshold* variable determines whether the two compared images are considered to be the same subject. The threshold is a value which denotes the maximum number of bits the NeuralHash can deviate from the anchor image's NeuralHash. Hence, the threshold is a limit for the maximum HamD the two compared NeuralHashes can have to be considered the same subject. The threshold variable is incremented by one after all images have been compared, in order to measure the FAR and FRR for all possible values of the threshold. The FAR and FRR are calculated according to 4.5 and 4.6 respectively. False Positive (FP) and True Negative (TN) are used to calculate FAR, while False Negative (FN) and True Positive (TP) are used to calculate FRR. Listing 4.7 shows how the FAR and FRR are implemented.

$$FAR = \frac{FP}{FP + TN} \quad (4.5)$$

$$FRR = \frac{FN}{FN + TP} \quad (4.6)$$

**Listing 4.7:** Pseudocode for calculating FAR and FRR. The complete code is listed in Appendix E

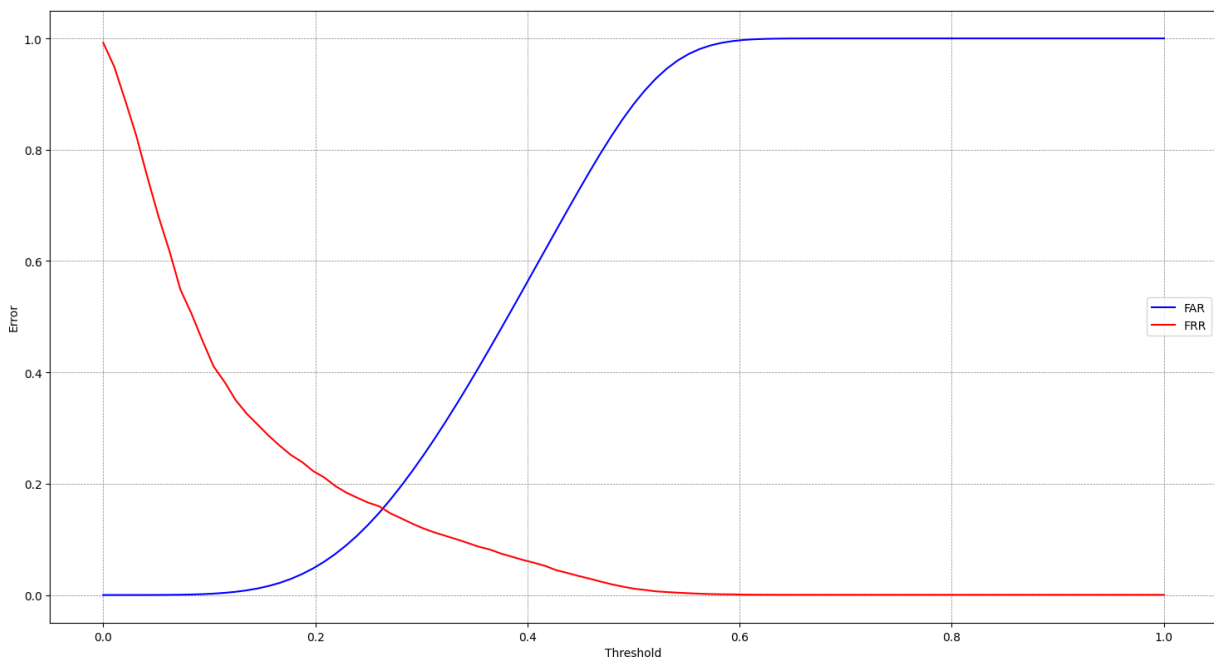
```

1 create empty lists of average_far and average_frr
2 for threshold in range(length of hash + 1):
3     for each subject:
4         calculate [true_positives, false_positives, true_negatives, false_negatives]
5         FAR = false_positives / (false_positives + true_negatives)
6         FRR = false_negatives / (false_negatives + true_positives)
7
8     append average FAR and FRR to lists for current threshold
9
10 plot average FAR and FRR

```

## 4.6 Results

Figure 4.7 compares the FAR and FRR for all possible values of HamD thresholds. The threshold along the x-axis ranges from 0 to 1, and denotes how similar two NeuralHashes must be for a match to be successful. A threshold of 0 means the NeuralHashes must be identical for a successful match, while a threshold of 1 means the match will always be successful. The error along the y-axis ranges from 0 to 1, where an FAR of 0 means no image comparisons are falsely categorised as the same subject, and an FRR of 0 means no image comparisons are falsely categorised as different subjects. Conversely, an FAR of 1 means all image comparisons of different subjects are falsely classified as the same subject, and an FRR of 1 means all image comparisons of the same subject are falsely classified as different subjects.



**Figure 4.7:** False Acceptance Rate and False Rejection Rate for varying HamD thresholds. The threshold denotes the maximum normalized HamD between two images.

Some points to note on the graph are that up to a threshold value of 0.1, almost no image comparisons of different subjects are classified as the same subject (only about 0.16 %). With a threshold value exceeding 0.6, almost all comparisons between different subjects



are falsely classified as the same subject (99.55 % at a threshold of 0.6). Further, with a threshold value exceeding 0.55, almost no image comparisons of the same subject are falsely categorised as different subjects (0.39 % at a threshold of 0.55). If the threshold is set to 0, meaning that the NeuralHashes must be identical to confirm a match, 99.08 % of image comparisons of the same subject are falsely rejected. The Equal Error Rate (EER) for the FAR and FRR is a threshold value of approximately 0.24. For this threshold, the FAR and FRR have a value of about 9.68 %.

## 4.7 Experiment 4

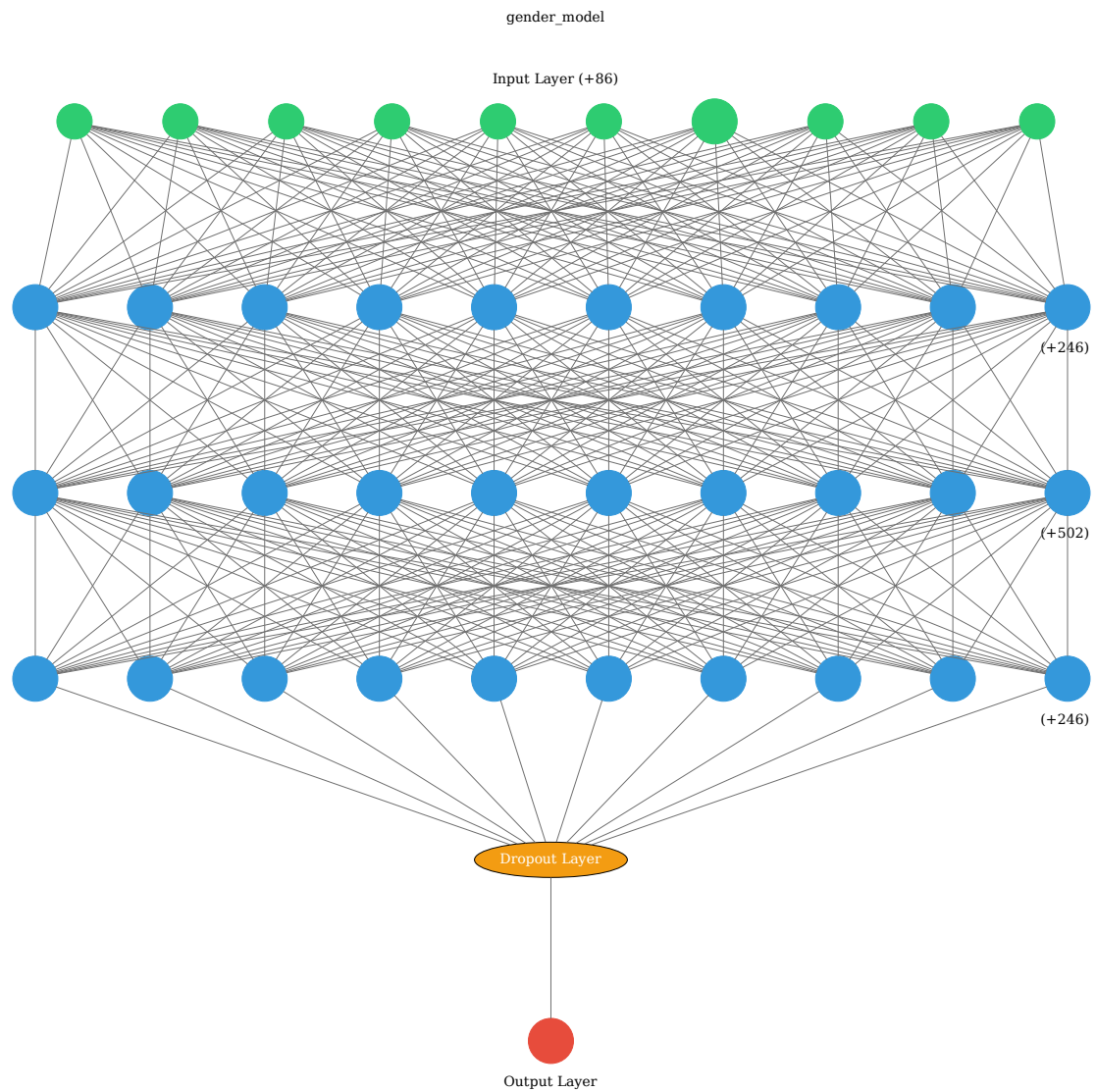
In the fourth experiment, the NeuralHash algorithm’s ability to hide information is examined. Two simple DFFNs are created in order to uncover any potential underlying patterns in the NeuralHashes of the various images. The DFFNs take a NeuralHash as input, and try to predict (1) the gender of the subject which the NeuralHash was generated from, and (2) the race of the subject which the NeuralHash was generated from.

### 4.7.1 Implementation

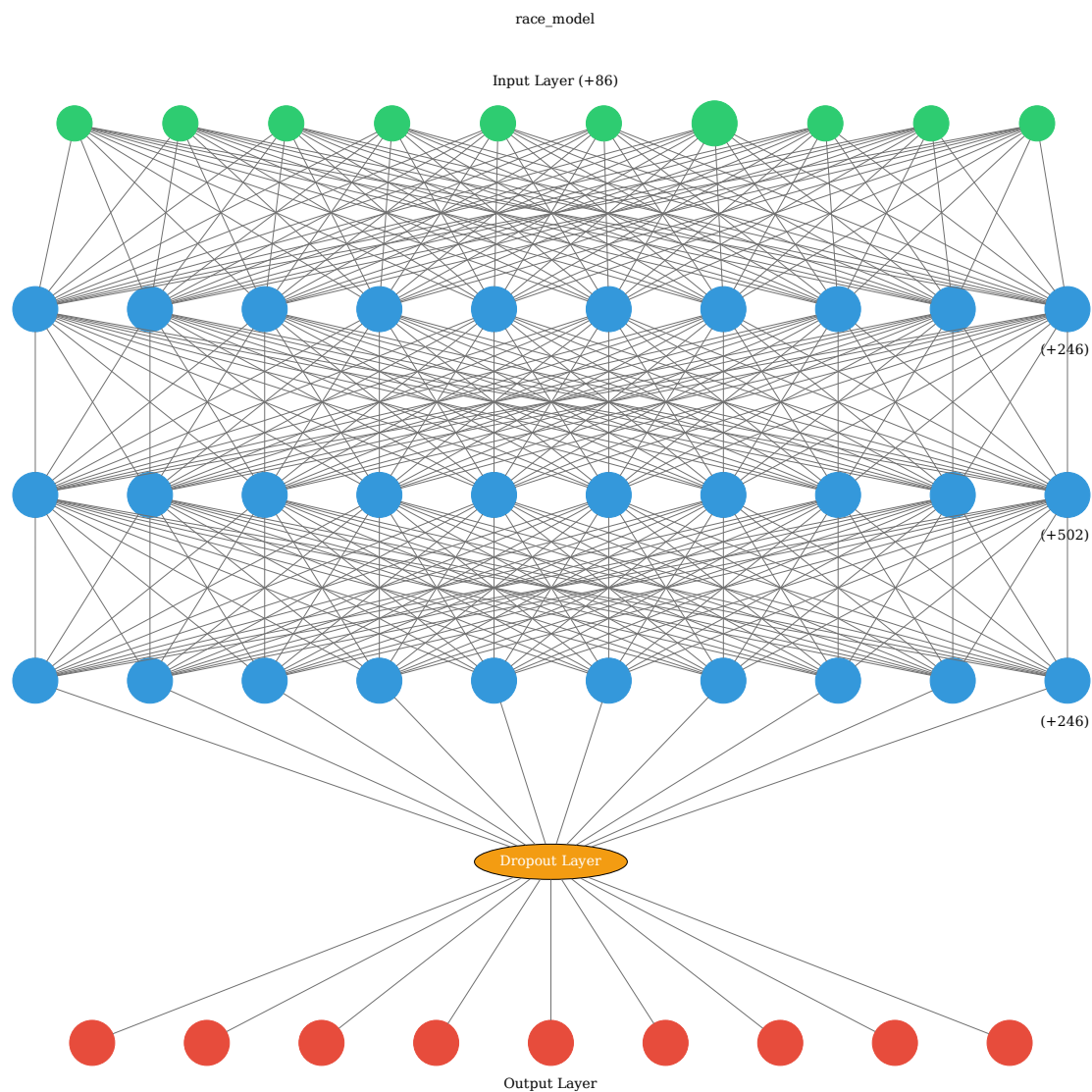
In Experiment 4, the labels for the images in the dataset come to use. A csv file is created containing information about the gender and race of every subject. Each row in the csv file contains the name of an image file, the corresponding binary NeuralHash, the gender, and the race of the subject in the image. This csv file is used to generate the dataset for the DFFNs used in this experiment. For both the DFFNs, only the NeuralHashes are used as input data for the networks. The labels for the gender network is the gender of the subject, and the labels for the race network is the race of the subject. In both cases, the dataset is shuffled and split into a training dataset and a testing dataset, and the split ratio is 0.7, meaning 70 % of the dataset is used for training, while 30 % is used for validation. However, since many of the NeuralHashes of the same subject are very similar, the data shuffling only shuffles subjects around, as opposed to individual images.

Hence, all images of a given subject end up either in the training dataset or the testing dataset. This is to prevent overfitting, where the models memorize the labels for certain NeuralHashes if some images of the corresponding subject end up in the training dataset while others end up in the testing dataset.

The DFFNs are simple networks taking the 96-bit NeuralHashes as input, and outputting a binary value for the gender network, and a one-hot encoded vector for the race network. The network structures are similar in the two cases, where the input layer has a size of 96 to accommodate for the bits in the NeuralHash, followed by dense layers of sizes 256, 512, 256, a dropout layer with a dropout rate of 0.2, and finally the output layer. The choice of this exact architecture for the networks is somewhat arbitrary, and could be optimized for better performance. The dropout layer is used to help prevent overfitting on the training data, as well as preventing co-adaption [30]. Figures 4.8 and 4.9 visualize the structures of the two networks, and Table 4.4 shows the hyperparameters used for training. The figures are created using the ANN-visualizer library for Python [31].



**Figure 4.8:** Structure of the gender network. The number next to each layer denotes how many additional neurons are present in the layer.



**Figure 4.9:** Structure of the race network. The number next to each layer denotes how many additional neurons are present in the layer.

	Gender model	Race model
Loss function	MSE	categorical cross-entropy
Optimizer	Adam	Adam
Epochs	150	150
Batch size	10	10
Validation split	0.3	0.3

**Table 4.4:** Hyperparameters for the models.

## 4.7.2 Results

Figures 4.10 and 4.12 show the training and validation accuracies for the gender and race models respectively. In order for the models to be useful, they need to exceed a baseline accuracy. Since the problem in this case is classification, the Zero Rate (ZeroR) Classifier can be used as a baseline. The ZeroR Classifier always predicts the label of the largest class [32]. In the case of the gender dataset, Table 4.2 shows that there are 1715 images of males, and 1007 images of females in the dataset, meaning 63.0 % of the dataset contains images of males. Hence, the ZeroR Classifier will always predict male, and consequently achieve an accuracy of 63.0 %. As for the Race dataset, the largest class is *White*, with 1691 out of 2722 images containing subjects of this class. This results in a ZeroR Classifier accuracy of 62.1 %.

The results from Experiment 4 show that the gender model is able to extract information from the NeuralHashes to predict the gender of the subject better than the ZeroR Classifier. The gender model has a validation accuracy of approximately 80 %, which is 17 % better than the ZeroR Classifier. This shows that the NeuralHashes contain some information about the images which can be retrieved using a Neural Network for binary classification.

The loss of the model is shown in Figure 4.11. The loss is relatively low for the validation, however, the discrepancy between the training and testing loss can indicate that the model is overfitting. This is even more evident when considering the discrepancy of accuracy in Figure 4.10, where the training accuracy is 100 %, while the validation accuracy is around 80 %.

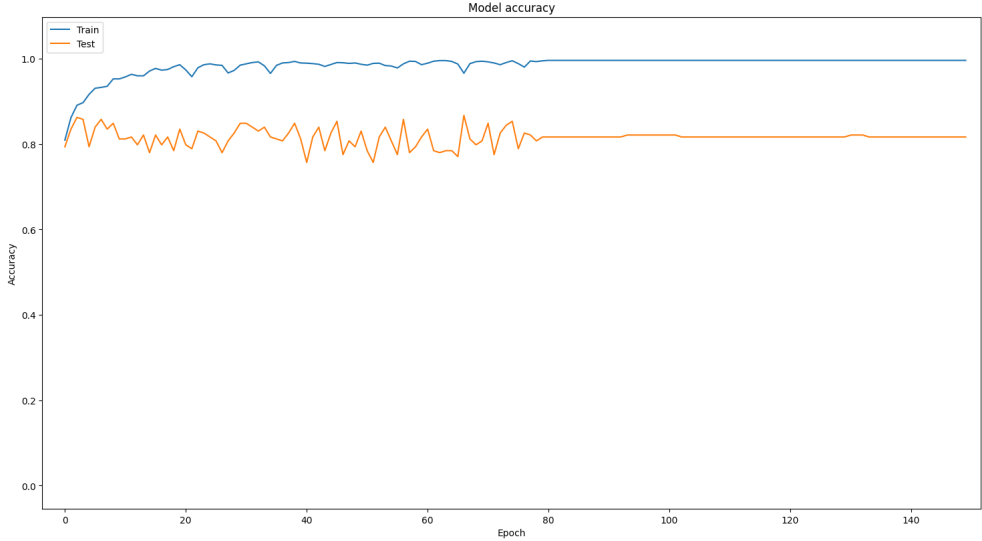


Figure 4.10: Training and validation accuracy values for the gender model.

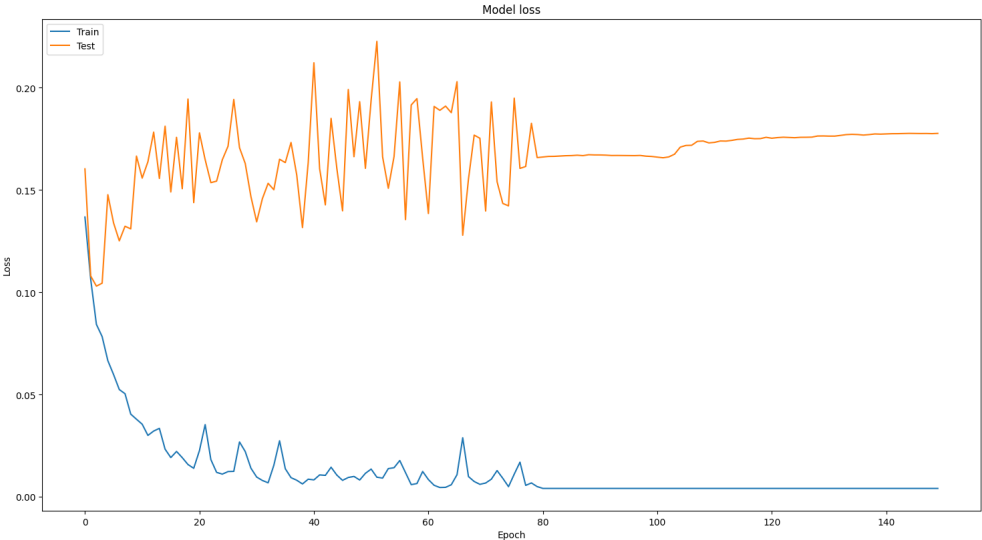
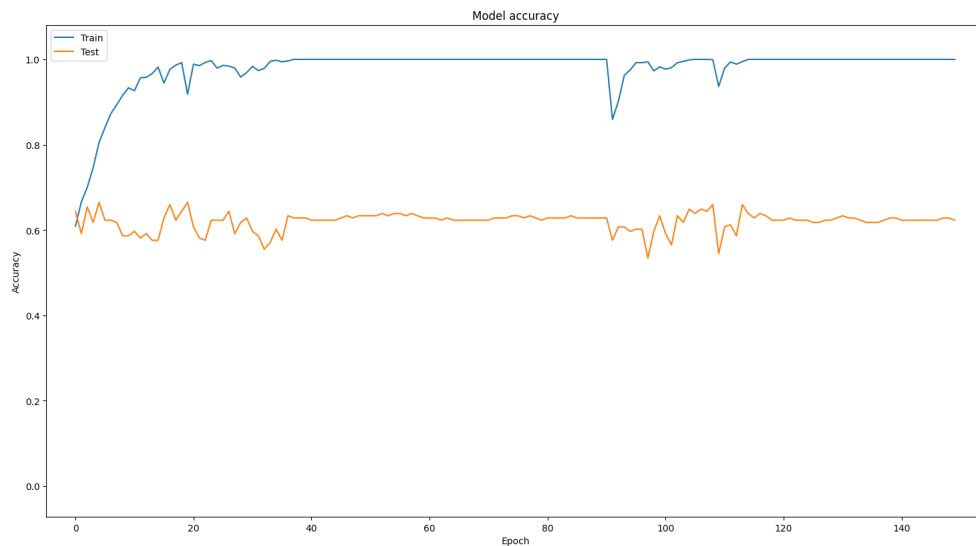


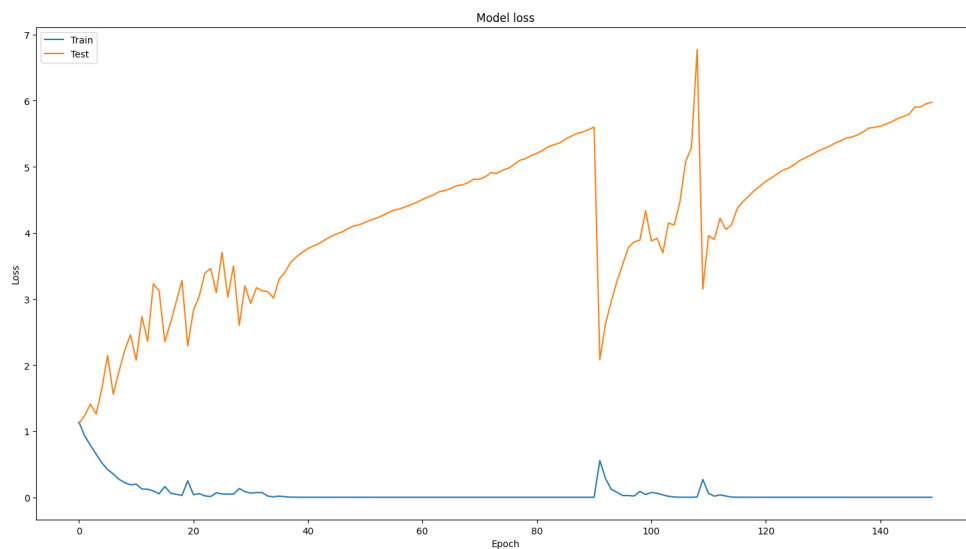
Figure 4.11: Training and validation loss values for the gender model.

The race model has a training accuracy of 100 %, and a validation accuracy of 60 %, as seen in Figure 4.12. This accuracy is lower than the ZeroR Classifier, meaning the model is unable to extract information about the race of the subjects from the NeuralHashes.

The loss of the model is shown in Figure 4.13, and shows a large discrepancy between the training and validation. This indicates the model is overfitting, and simply memorizing the input-output pairs from the training samples. This is likely the reason why the training accuracy and loss values are good, while the validation values are bad.



**Figure 4.12:** Training and validation accuracy values for the race model.



**Figure 4.13:** Training and validation loss values for the race model.

In addition to the experiments above, two additional experiments are implemented; Experiment 5 and 6. These experiments are *Adversary 1 - Hash Collision Attacks* and *Adversary 2 - Gradient-Based Evasion Attacks* in the paper by Struppek et al. (2022) [13], using images from the ColorFERET Dataset. The code for the experiments by Struppek et al. is available on GitHub at [33].

## 4.8 Experiment 5

In Experiment 5, a small subset of subjects,  $S$ , was selected for image manipulation. Next, a surrogate hash database containing images of all subjects in the ColorFERET Dataset except the subjects in  $S$  was created. Next, the `adv1_collision_attack.py` file was invoked with images of the subjects in  $S$ , and the surrogate hash database as input. The output images were visually indistinguishable from the originals, and the NeuralHashes of the altered images matched a different subject's image exactly.

### 4.8.1 Results

Table 4.5 shows the input images for the `adv1_collision_attack.py` script in the top row, output images in the middle row, and target subjects in the bottom row. The NeuralHashes of the original images are displayed directly beneath the corresponding images, and the NeuralHashes for both the altered images, and images of the target subjects are displayed at the bottom. The NeuralHashes for both the altered image, and the target subject match exactly for all four subjects. The NeuralHashes are here represented in hexadecimal, and the differences in the NeuralHashes are highlighted in red.



original				
	853573f622dc66f548b368a2	b7270ab78a64465338360c2a	4068b1046eea6ba825bb82a2	8921b37873bffe4449afc28a
altered				
target subject				
	853571f0229166d548b360ca	33270bab8264741324360c02	5068b1476ba69ec24ab1282	8921b16873bffe4449afc28a

Table 4.5: Results from Adversary 1.

## 4.9 Experiment 6

Experiment 6 is based on altering an input image such that the NeuralHash differs from the original image, while keeping the visual impact of the changes at a minimum. A parameter, *hamming* (here denoted as  $h$ ), denotes proportionally how different the NeuralHash must be before the output image is accepted, where  $h$  is in the range  $[0, 1]$ . The *adv2\_evasion\_attack.py* and a modified version of *adv2\_few\_pixels\_attack.py* scripts were used to run the experiment with various values for  $h$ .

The modified *adv2\_few\_pixels\_attack.py* script can be found in Appendix F.

### 4.9.1 Results

Table 4.6 shows the outputs from experiment 6 for subject 00407. The top image is the original input image. The **standard** row refers to the standard evasion attack described in the *Adversary 2 - Gradient-Based Evasion Attacks* section of [13], and the images were generated by running the commands in Listing 4.8. The **edges\_only** row was generated by appending the `-edges_only` flag to the commands, as shown in Listing 4.9. The last row containing the image from the **few\_pixels** attack was generated by running the command shown in Listing 4.10. The *hamming* or *h* value denotes how much the binary representation of the NeuralHash must change before the output image is accepted. Hence, the images in the table have NeuralHashes which have been changed by at least 1 %, 10 % and 50 %.

All the attack types were run with *h* values of 0.01, 0.1, and 0.5. However, the *edges\_only* and *few\_pixels* attack variants were unable to produce a result with an *h* value of 0.5. The *few\_pixels* attack could not generate an output with an *h* value of 0.1 either, as the limit of 150 pixels was exceeded before the hash was changed sufficiently. Each image also has an attached NeuralHash below, where the differences from the original image’s NeuralHash is highlighted in red.

**Listing 4.8:** Commands for generating images using the standard evasion attack.

```

1 python3 adv2_evasion_attack.py --hamming=0.01 --source=inputs/adv2_input --output_folder=
  ↪ outputs/adv2
2 python3 adv2_evasion_attack.py --hamming=0.1 --source=inputs/adv2_input --output_folder=
  ↪ outputs/adv2
3 python3 adv2_evasion_attack.py --hamming=0.5 --source=inputs/adv2_input --output_folder=
  ↪ outputs/adv2

```

**Listing 4.9:** Command for generating images using the `edges_only` attack.








```

1 python3 adv2_evasion_attack.py --hamming=0.01 --edges_only --source=inputs/adv2_input --
  ↪ output_folder=outputs/adv2

```

**Listing 4.10:** Command for generating images using the few\_pixels attack

```
1 python3 adv2_few_pixels_attack.py --hamming=0.01 --source=inputs/adv2_input --
  ↪ output_folder=outputs/adv2
```

original	 4068b1046eea6ba825bb82a2		
	<b>h=0.01</b>	<b>h=0.1</b>	<b>h=0.5</b>
standard	 4068b1046eea6ba825bb82a2	 4160b1006effe6ba825bb86a2	 ed2f5802852fea24dd2daf2c
edges_only	 4068b1046eea6ba825bb82a2	 4160b1006effe6ba825bb86a2	N/A
few_pixels	 4068b1046eea6ba825bb82a2	N/A	N/A

**Table 4.6:** Results from Adversary 2.

Both the images for  $h=0.01$  and  $h=0.1$  in the standard evasion attack are visually hard to distinguish from the original. However, the image for  $h=0.5$  contains some clearly visible diagonal stripes. This is not surprising, given how much the NeuralHash has changed from the original. The *edges\_only* attack with  $h=0.01$ , again, looks indistinguishable from the original. However, when  $h=0.1$  for this attack, the output image contains some ugly perturbations across the face. As for the *few\_pixels* attack, the image again looks identical to the original. The only change in this image are three pixels on the bottom right, which have changed the NeuralHash by one bit.

## 5 | Discussion

In this chapter, the thesis in its entirety is discussed. The experimental results are used to answer the RQs defined in Section 1.4. The hypothetical scenario defined in Section 3.1 is evaluated in context of the results. Finally, the future work for the thesis is presented.

As mentioned in the Related work Section, Google's FaceNet was able to achieve a classification accuracy of 99.63 %  $\pm$ 0.09. By using a threshold matching the EER for the system presented in this thesis, NeuralHash can be used to both correctly match the same individual, and correctly classify different individuals as different in 90.32 % of cases. It is clear that FR algorithms have better performance when they have access to facial images directly. Furthermore, the model used in FaceNet was trained on up to 260,000,000 facial images [6], while the amount and type of training data for the NeuralHash model is unknown.

Microsoft's PhotoDNA algorithm is strikingly similar to Apple's NeuralHash algorithm in their functionality. Microsoft states that "*PhotoDNA is not facial recognition software and cannot be used to identify a person or object in an image*" [14]. However, since it is used to detect similar images, it would not be surprising if it could be used in an application such as the one presented in this thesis. Furthermore, Microsoft states that "*A PhotoDNA hash is not reversible, and therefore cannot be used to recreate an image*" [14]. If this claim is true, PhotoDNA is a promising candidate for comparing against NeuralHash in a FR scenario. The only problem is that the source code for PhotoDNA is proprietary. If the source code becomes publicly available in the future, it could be a contender for NeuralHash in the system implemented in this thesis.

The ColorFERET Dataset used in this thesis present facial images of subjects with varying facial expressions and image cropping. Some images are cropped to fit only the face,

while others feature large portions of the subject’s torso. Many subjects have multiple images from the same photo session, where the face position in the image and clothing of the subject stay consistent. These types of images often generate similar NeuralHashes. However, a significant portion of the features used to generate the NeuralHash depend on the clothing, background and other features in the image beside the subjects’ faces. Hence, an attempt was made to crop the dataset images to only contain the faces of the subjects. However, the performance on this dataset was poor, and not included in the thesis. This limitation may be circumvented by performing transfer learning, or by preprocessing the images differently.

As demonstrated by Brad Dwyer, natural collisions occur in NeuralHash [34]. As such, it is not unthinkable that facial images of different people can result in the same NeuralHash. This can be an advantage for the proposed system in terms of privacy, as the processed NeuralHashes cannot be mapped directly to an identity. Furthermore, variations in facial expression, lighting and other factors can cause the NeuralHash of a facial image to vary considerably. Hence, a cluster of neighboring NeuralHashes can be mapped to the same person, which causes even more overlaps with other identities.

## 5.1 Experimental results summary

Experiment 1 has shown a method for identifying whether two facial images belong to the same person. The results show that images of the same subject in general have a lower HamD than images of different subjects. Consequently, the answer to RQ 1 is that NeuralHashes representing facial images can be used to identify whether two facial images belong to the same person by measuring the HamD between the NeuralHashes.

Experiment 2 demonstrates how reliable NeuralHash is in clustering images of the same subject close together. The results show that NeuralHash consistently clusters images of the same subject closer together than images of different subjects, and that the SelfDists are more spread out than the DiffDists.

Experiment 3 demonstrates a technique for measuring how consistently NeuralHash classifies identities correctly by measuring the FAR and FRR for varying HamD thresholds. The results shows that NeuralHash classifies identities correctly most of the time, however, there is room for improvement.

Experiment 2 and 3 together demonstrate a method for evaluating the classification performance of NeuralHash. The reliability can be measured by calculating and comparing the average SelfDist and DiffDist, in addition their standard deviations. The accuracy can be evaluated by varying the HamD threshold and calculating the FAR and FRR for each threshold. Together, these techniques provide a method for evaluating the classification performance.

The results from Experiment 4 show that a NeuralHash can reveal some non-specific information about the input image, such as gender of the subject. However, when the classes become more nuanced, the Neural Networks struggle to consistently predict the correct class. The loss of the race model evidently shows that the model is very insecure in its predictions, meaning a given prediction has limited credibility. The answer to RQ 3 is therefore that it is possible to retrieve some general information from facial images such as gender, however, more nuanced information is difficult, if not impossible to retrieve.

## 5.2 Hypothetical scenario

Regarding the hypothetical scenario introduced in Section 3.1, Experiment 3 shows that if the threshold is set to the EER at 0.24, the system will fail to recognize 9.68 % of persons of interest, and falsely match 9.68 % of ordinary people as persons of interest. Since a positive match involves additional, potentially manual security measures, it is best to reduce the number of false positives at the cost of increasing false negatives. The rationale for this is that the result experiments are based on a single matching attempt for each image in the database. If the system is scaled up to process a handful of images for each person instead of just one, the probability of a false negative is lowered to  $p^n$ , where  $p$  is

the probability of a false negative, and  $n$  is the number of images processed. This assumes that the images capture somewhat different features, meaning they are not monotonous. According to the results, if the threshold is set to 0.1, only 0.16 % of ordinary people will be falsely matched, while 31.45 % of persons of interest will be falsely ignored. This could be an acceptable trade-off.

One critical point about the ColorFERET Dataset, and results presented in this thesis in relation to the hypothetical scenario is that many of the images of the subjects in the dataset are captured in rapid succession. In the hypothetical scenario, the database of persons of interest most certainly contains dated images where the individuals have different appearance due to aging and everything that entails. As such, it is necessary to further investigate how NeuralHash performs with this kind of data to provide practical results for such a scenario.

One potential problem with using NeuralHash to match facial images is that the database containing NeuralHashes of persons of interest is tied to a particular version of the NeuralHash model. It is possible that the NeuralHash model could see improvements over time through additional training, however, this could lead to faces being mapped to completely new hash values. As such, the database would have to be updated along with the NeuralHash model, and some version tracking system would be necessary.

The results from Experiment 5 and 6 show that it is possible to inconspicuously alter facial images such that the NeuralHash either matches a target hash value, or such that the NeuralHash differs from the original image by a certain threshold. This could be very problematic if the system relies on pre-captured images, such as passport images, as these can be altered by an adversary beforehand. However, since the system relies on capturing new images of each person, there is no possibility to alter the images before they are analysed by the system.

In the hypothetical scenario, it is assumed that the database containing persons of interest is stored securely, with no unauthorized access or modification. However, if an adversary



was able to insert new NeuralHashes into the database, the adversary could harass a victim by inserting a NeuralHash corresponding to the victim.

### 5.3 Future work

The hyperplanes defined in the *neuralhash\_128x96\_seed1.dat* file define the limits for which feature vectors are mapped to a 0, and which are mapped to a 1. Apple does not specify how these particular hyperplanes are selected, or if they are optimized to improve the ability of NeuralHash to assign similar images the same hash value. If these hyperplanes are defined randomly, there could be potential to improve NeuralHash’s ability to use features in facial images to assign hash values, by optimizing the hyperplanes.

The NeuralHash algorithm is a fully trained deep perceptual hashing algorithm created and used by Apple Inc. in their products. The neural network and its parameters are available in the iOS operating system, however, the data, loss functions and hyperparameters used to train the network are not [13]. The network has been trained to recognize similar images, although it is uncertain how well it has been trained to recognize faces specifically. If it is possible to train the network to better recognize faces, the performance of the system presented in this thesis could be improved. One way to specialize a model is with transfer learning. This technique can be used to initialize a model with prior knowledge, and use that knowledge to speed up the process of learning specialized skills. The NeuralHash model could be used to initialize a new model with the knowledge of general image similarity, and further trained to pick up more specific features in facial images.

## 6 | Conclusion

In this thesis, a system for identity matching using the NeuralHash algorithm was presented. The goal was to implement the system, and evaluate its privacy and security aspects. The NeuralHash algorithm was used to generate fingerprints of subjects in a subset of the ColorFERET facial image Dataset, and the HamD between these fingerprints was used to match subjects without revealing the facial images. The FAR and FRR was measured for different thresholds, and two simple DFFNs were implemented to analyse the NeuralHash algorithm's ability to hide information. Additionally, two experiments were implemented to investigate hash collisions in NeuralHash, as well as hash altering. The results show that NeuralHash can be used to identify whether two facial images belong to the same person. The classification can be somewhat inconsistent, for example if the threshold for maximum deviation in the NeuralHash is set to 10 %, the FAR is 0.16 %, however, the FRR is 31.45 %. NeuralHash also hides the image information sufficiently for the purposes of FR. It is possible to retrieve some general information from facial images such as gender, however, more nuanced information is difficult, if not impossible to retrieve.

# References

- [1] P. J. Phillips, H. Wechsler, J. Huang, and P. Rauss, “The feret database and evaluation procedure for face recognition algorithms,” *Image and Vision Computing*, vol. 16, no. 5, pp. 295–306, 1998.
- [2] P. J. Phillips, H. Moon, S. A. Rizvi, and P. J. Rauss, “The feret evaluation methodology for face recognition algorithms,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 22, pp. 1090–1104, 2000.
- [3] PapersWithCode, “Face recognition.” <https://paperswithcode.com/task/face-recognition>. Accessed: Apr. 1, 2022.
- [4] W. W. Bledsoe, “A facial recognition project report.” <https://archive.org/details/firstfacialrecognitionresearch/FirstReport/page/n33/mode/2up>, 1963. Accessed: Apr. 1, 2022.
- [5] J. Deng, J. Guo, N. Xue, and S. Zafeiriou, “Facenet: A unified embedding for face recognition and clustering.” <https://arxiv.org/pdf/1801.07698v3.pdf>, 2019. Accessed: Apr. 1, 2022.
- [6] F. Schroff, D. Kalenichenko, and J. Philbin, “Facenet: A unified embedding for face recognition and clustering.” <https://arxiv.org/pdf/1503.03832.pdf>, 2015. Accessed: Apr. 1, 2022.
- [7] Y. Sun, X. Wang, and X. Tang, “Deep learning face representation by joint identification-verification.” <https://arxiv.org/pdf/1406.4773v1.pdf>, 2014. Accessed: Apr. 1, 2022.
- [8] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “Deepface: Closing the gap to human-level performance in face verification.” [https://www.cs.toronto.edu/~ranzato/publications/taigman\\_cvpr14.pdf](https://www.cs.toronto.edu/~ranzato/publications/taigman_cvpr14.pdf), 2014. Accessed: Apr. 1, 2022.

- [9] Apple, “Face id security.” [https://www.apple.com/business-docs/FaceID\\_Security\\_Guide.pdf](https://www.apple.com/business-docs/FaceID_Security_Guide.pdf), 2017. Accessed: May 6, 2022.
- [10] R. Ullah, H. Hayat, A. A. Siddiqui, U. A. Siddiqui, J. Khan, F. Ullah, S. Hassan, L. Hasan, W. Albattah, M. Islam, and G. M. Karami, “A real-time framework for human face detection and recognition in cctv images.” <https://downloads.hindawi.com/journals/mpe/2022/3276704.pdf>, 2022. Accessed: May 7, 2022.
- [11] “Art. 4 gdpr definitions.” <https://gdpr-info.eu/art-4-gdpr/>, 2016. Accessed: Apr. 29, 2022.
- [12] Apple Inc., “Csam detection technical summary.” [https://www.apple.com/child-safety/pdf/CSAM\\_Detection\\_Technical\\_Summary.pdf](https://www.apple.com/child-safety/pdf/CSAM_Detection_Technical_Summary.pdf), 2021. Accessed: May 12, 2022.
- [13] L. Struppek, D. Hintersdorf, D. Neider, and K. Kersting, “Learning to break deep perceptual hashing: The use case neuralhash.” <https://arxiv.org/pdf/2111.06628.pdf>, 2022. Accessed: Apr. 6, 2022.
- [14] Microsoft, “Photodna.” <https://www.microsoft.com/en-us/photodna>, 2015. Accessed May 31, 2022.
- [15] Microsoft Research, “Photodna™: How it works.” <https://www.youtube.com/watch?v=NOR1SXfcW1o>, 2009. Accessed May 31, 2022.
- [16] T. Ferrandez, “Paper reviews call 002 – facenet: A unified embedding for face recognition and clustering.” <https://www.youtube.com/watch?v=w--c0qG9MCc>, 2019. Accessed May 31, 2022.
- [17] A. Ziller, J. Passerat-Palmbach, T. Ryffel, D. Usynin, A. Trask, I. Da Lima Costa Junior, J. Mancuso, M. Makowski, D. Rueckert, R. Braren, and G. Kaissis, “Privacy-preserving medical image analysis.” <https://arxiv.org/pdf/2012.06354.pdf>, 2020. Accessed: May 12, 2022.
- [18] Wikipedia, “Perceptual hashing.” [https://en.wikipedia.org/wiki/Perceptual\\_hashing](https://en.wikipedia.org/wiki/Perceptual_hashing), 2022. Accessed: May 27, 2022.

- [19] Wikipedia, “Locality-sensitive hashing.” [https://en.wikipedia.org/wiki/Locality-sensitive\\_hashing](https://en.wikipedia.org/wiki/Locality-sensitive_hashing), 2022. Accessed: May 27, 2022.
- [20] D. Steen, “A gentle introduction to self-training and semi-supervised learning.” <https://towardsdatascience.com/a-gentle-introduction-to-self-training-and-semi-supervised-learning-ceed73178b38>, 2020. Accessed May 28, 2022.
- [21] M. Bekuzarov, “Losses explained: Contrastive loss.” <https://medium.com/@maksym.bekuzarov/losses-explained-contrastive-loss-f8f57fe32246>, 2020. Accessed May 28, 2022.
- [22] Wikipedia, “Hamming distance.” [https://en.wikipedia.org/wiki/Hamming\\_distance](https://en.wikipedia.org/wiki/Hamming_distance), 2021. Accessed: Mar. 21, 2022.
- [23] NIST, “Face recognition technology (feret).” <https://www.nist.gov/programs-projects/face-recognition-technology-feret>, 2019. Accessed: Mar. 23, 2022.
- [24] Wikipedia, “bzip2.” <https://en.wikipedia.org/wiki/Bzip2>, 2022. Accessed: Mar. 23, 2022.
- [25] A. Ygvar, “Appleneuralhash2onnx.” <https://github.com/AsuharietYgvar/AppleNeuralHash2ONNX>, 2021. Accessed: Mar. 24, 2022.
- [26] IPSW, “ios 15.2.1 (19c63) for iphone 13.” <https://ipsw.me/download/iPhone14,5/19C63>, 2022. Accessed: Mar. 24, 2022.
- [27] lzsfe, “Lzsfe.” <https://github.com/lzfse/lzfse>, 2017. Accessed: Mar. 24, 2022.
- [28] Tencent Youtu Lab and Guangying Lab, “TNN.” <https://github.com/AsuharietYgvar/TNN>, 2021. Accessed: Mar. 24, 2022.
- [29] A. Haapala, “python-levenshtein 0.12.2.” <https://pypi.org/project/python-Levenshtein/>, 2021. Accessed: Apr. 5, 2022.

- 
- [30] Nitin1901, “Dropout in neural networks.” <https://www.geeksforgeeks.org/dropout-in-neural-networks/>, 2020. Accessed: May 22, 2022.
- [31] T. Gheorghiu, “Ann visualizer.” <https://github.com/RedaOps/ann-visualizer>, 2018.
- [32] A. Lee, “Choosing a baseline accuracy for a classification model.” <https://towardsdatascience.com/calculating-a-baseline-accuracy-for-a-classification-model-a4b342ceb88f>, 2021. Accessed May 24, 2022.
- [33] L. Struppek, D. Hintersdorf, D. Neider, and K. Kersting, “Learning to break deep perceptual hashing: The use case neuralhash (facct 2022).” <https://github.com/ml-research/Learning-to-Break-Deep-Perceptual-Hashing>, 2022. Accessed: May 28, 2022.
- [34] B. Dwyer, “neuralhash-collisions.” <https://github.com/roboflow-ai/neuralhash-collisions>, 2021. Accessed: May 30, 2022.

# A | Data cleaning

Listing A.1 shows the python module for cleaning the ColorFERET Dataset.

**Listing A.1:** Removing all non-frontal face images

```
1 import os
2
3
4 if __name__ == "__main__":
5
6     img_dir = f"{os.getcwd()}/images"
7     for subject_dir_name in os.listdir(img_dir):
8         print(f"{subject_dir_name}")
9         subject_dir = f"{img_dir}/{subject_dir_name}"
10        for img_file in os.listdir(subject_dir):
11            file_path = f"{subject_dir}/{img_file}"
12            if 'f' not in img_file:
13                os.remove(file_path)
```

## B | Inference

Listing B.1 shows the NeuralHash class. The `calculate_neuralhash` method can be invoked with an input image to retrieve the NeuralHash of the image.

**Listing B.1:** Code for the NeuralHash class used for inference.

```
1 class NeuralHash:
2     def __init__(self):
3         # Load ONNX model
4         self.model_path = f"{os.getcwd()}/model/model.onnx"
5         self.session = InferenceSession(self.model_path)
6
7         # Load output hash matrix
8         seed1_path = f"{os.getcwd()}/model/neuralhash_128x96_seed1.dat"
9         self.seed1 = open(seed1_path, 'rb').read()[128:]
10        self.seed1 = np.frombuffer(self.seed1, dtype=np.float32)
11        self.seed1 = self.seed1.reshape([96, 128])
12
13    def calculate_neuralhash(self, image_path):
14        """Calculate neuralhash of the image at image_path"""
15
16        arr = self.im2array(image_path)
17
18        # Run model
19        inputs = {self.session.get_inputs()[0].name: arr}
20        outs = self.session.run(None, inputs)
21
22        # Convert model output to hex hash
23        hash_output = self.seed1.dot(outs[0].flatten())
24        hash_bits = ''.join(['1' if it >= 0 else '0' for it in hash_output])
25        hash_hex = '{:0{x}}'.format(int(hash_bits, 2), len(hash_bits) // 4)
26
27        return hash_hex, hash_bits
28
29    @staticmethod
30    def im2array(image_path):
31        """Preprocess image"""
32
33        image = Image.open(image_path).convert('RGB')
34        image = image.resize([360, 360])
```



```
35     arr = np.array(image).astype(np.float32) / 255.0
36     arr = arr * 2.0 - 1.0
37
38     return arr.transpose(2, 0, 1).reshape([1, 3, 360, 360])
```

---

# C | Calculate Hamming Distances

Listing C.1 shows the method used for calculating and storing Hamming Distances for all images in the *self.hash\_dict* dictionary.

**Listing C.1:** Method for calculating and storing Hamming Distances.

```
1 def calculate_hamming_distances(self, max_comp=None):
2     """
3     Calculate hamming distances of NeuralHashes for
4     each image pair and store in self.hamming_distances,
5     or only compare up to max_comp if supplied
6     """
7
8     if not max_comp:
9         max_comp = len(self.hash_dict)
10
11    for n, i in enumerate(self.hash_dict):
12        dict_slice = dict(itertools.islice(self.hash_dict.items(), n+1, max_comp))
13        for j in dict_slice:
14            hamming_dist_hex = Levenshtein.hamming(self.hash_dict[i][0],
15                                                    self.hash_dict[j][0])
16            hamming_dist_bin = Levenshtein.hamming(self.hash_dict[i][1],
17                                                    self.hash_dict[j][1])
18            subject_i = i.split(self.split_char)[0]
19            subject_j = j.split(self.split_char)[0]
20            same = subject_i == subject_j
21
22            if i in self.hamming_distances:
23                self.hamming_distances[i][j] = (same, hamming_dist_hex, hamming_dist_bin)
24            else:
25                self.hamming_distances[i] = {j: (same, hamming_dist_hex, hamming_dist_bin
↪ )}
26
27            if j in self.hamming_distances:
28                self.hamming_distances[j][i] = (same, hamming_dist_hex, hamming_dist_bin)
29            else:
30                self.hamming_distances[j] = {i: (same, hamming_dist_hex, hamming_dist_bin
↪ )}
```

# D | Average Hamming Distances and Standard Deviation

Listing D.1 shows the method used for calculating and outputting average HamDs and standard deviations for each subject.

**Listing D.1:** Method for calculating average HamD and standard deviation for each subject.

```
1 def print_avg_hamming_distance_and_sd(self, sorting=DIST_SAME):
2     """
3     For each subject, print average hamming distance
4     to other images of the same subject, and average
5     hamming distance to images of other subjects.
6     Also print standard deviation of hamming distance
7     between images of the same subject and different
8     subjects.
9     :param sorting: int
10         Determines how to sort the output
11         DIST_SAME sorts on hamming distance between images
12         of same subject, from lowest to highest
13         DIST_OTHERS sorts on hamming distance between
14         different subjects, from lowest to highest
15         SD_SAME sorts on standard deviation for images
16         of the same subject, from lowest to highest
17         SD_OTHERS sorts on standard deviation for images
18         of different subjects, from lowest to highest
19     """
20
21     if sorting not in [DIST_SAME, DIST_OTHERS, SD_SAME, SD_OTHERS]:
22         print(f"{RED}Unexpected value for parameter \"sorting\"\n"
23               f"Expected int in range [0, 3]\n"
24               f"DIST_SAME = 0\n"
25               f"DIST_OTHERS = 1\n"
26               f"SD_SAME = 2\n"
27               f"SD_OTHERS = 3\n"
28               f"Ex: sorting=SD_SAME{END}")
29     return
30
```

```

31     avg_dist = {}
32     sorted_avg_dist_sd = {}
33     for file_i, dict_i in self.hamming_distances.items():
34         subject_i = file_i.split(self.split_char)[0]
35         if subject_i not in avg_dist:
36             avg_dist[subject_i] = [0, 0, 0, 0] # [sum_dist_same, sum_dist_different,
↪ total_same, total_different]
37
38         for file_j, dist in dict_i.items():
39             # Add hex or bin distance from file_i to file_j to sum_dist_same or
↪ sum_dist_different respectively
40             avg_dist[subject_i][not dist[0]] += dist[1 + self.output_format]
41             # Increment total_same or total_different respectively
42             avg_dist[subject_i][(not dist[0]) + 2] += 1
43
44     for subject, tally in avg_dist.items():
45         try:
46             # (sum_dist_same / total_same, sum_dist_different / total_different)
47             sorted_avg_dist_sd[subject] = [tally[0] / tally[2], tally[1] / tally[3]]
48         except (IndexError, ZeroDivisionError) as e:
49             print(f"{RED}{e} for subject {subject}{END}")
50
51     # Build lists of (hamming distance - average hamming distance) ** 2 for each
↪ comparison and store in deviation
52     deviation = {}
53     for file_i, dict_i in self.hamming_distances.items():
54         subject = file_i.split(self.split_char)[0]
55         if subject not in deviation:
56             deviation[subject] = [[], []] # [[distance same], [distance different]]
57         for file_j, dist in dict_i.items():
58             try:
59                 deviation[subject][not dist[0]].append((dist[1 + self.output_format]
60                 - sorted_avg_dist_sd[subject][not
↪ dist[0]]) ** 2)
61             except KeyError:
62                 pass
63
64     # Calculate standard deviation of hamming distances for same and different subjects
65     for subject, dev in deviation.items():
66         try:
67             variance_same = sum(dev[0]) / len(dev[0])
68             variance_different = sum(dev[1]) / len(dev[1])
69             sd_same = math.sqrt(variance_same)
70             sd_different = math.sqrt(variance_different)
71             sorted_avg_dist_sd[subject] += [sd_same, sd_different]
72         except ZeroDivisionError:

```

```

73         pass
74
75     # Sort by value determined by sorting
76     sorted_avg_dist_sd = dict(sorted(sorted_avg_dist_sd.items(), key=lambda x: x[1][
↵ sorting]))
77
78     # Print output to a grid
79     print(self.hl)
80     column_titles = "{0}    {1}    {2}    {3}".format("Avg dist self", "\u03C3 self", "
↵ Avg dist others",
81
82                                     "\u03C3 others")
83     print("{0:>{x}}    {1}".format("", column_titles, x=len(list(avg_dist.items())[0][0])
↵ + 1))
84
85     for subject, data in sorted_avg_dist_sd.items():
86         dist_self = "{dist:.2f}".format(dist=(data[DIST_SAME]))
87         dist_others = "{dist:.2f}".format(dist=(data[DIST_OTHERS]))
88         sd_same = "{sd:.2f}".format(sd=data[SD_SAME])
89         sd_others = "{sd:.2f}".format(sd=data[SD_OTHERS])
90         print("{0:>{x}}    {1:>13}    {2:>6}    {3:>15}    {4:>8}".format(
91             f"{subject}:", dist_self, sd_same, dist_others, sd_others, x=len(subject) + 1
92             ))

```

# E | FAR & FRR

Listings E.1, E.2, E.3, E.4 and E.5 show the methods involved in plotting the FAR and FRR for varying threshold values.

**Listing E.1:** Method for aggregating the necessary data to plot average FAR and FRR.

```
1 def plot_far_frr(self):
2     """
3     Plot the average FAR and FRR for threshold
4     varying from 0 to length of hash
5     """
6
7     avg_FAR = []
8     avg_FRR = []
9     for i in range(self.get_hash_length()+1):
10        self.calculate_rates(threshold=i)
11        FAR_i, FRR_i = self.calculate_avg_error_rate()
12        avg_FAR.append(FAR_i)
13        avg_FRR.append(FRR_i)
14        print(f"FAR_{i}: {FAR_i}, FRR_{i}: {FRR_i}")
15
16    self.plot_avg_error(avg_FAR, avg_FRR)
```

**Listing E.2:** Method for calculating error rates.

```
1 def calculate_rates(self, threshold=None):
2     """Calculate FAR and FRR for each subject"""
3
4     if not threshold:
5         threshold = self.threshold
6
7     # {subject: [true_positives, false_positives, true_negatives, false_negatives, FAR,
8     ↪ FRR, image, {img_files}]}
9     self.rates = {}
10    for image in self.hamming_distances:
11        subject = image.split(self.split_char)[0]
12        accepted, rejected, genuine, imposters = self.get_accepted_rejected(image,
13        ↪ threshold)
14        true_positives = accepted[0]
```

```

13     false_positives = accepted[1]
14     true_negatives = rejected[0]
15     false_negatives = rejected[1]
16
17     if true_positives > 0:
18         FAR = false_positives / (false_positives + true_negatives)
19         FRR = false_negatives / (false_negatives + true_positives)
20         if subject in self.rates:
21             if self.rates[subject][4] > FAR:
22                 self.rates[subject] = [true_positives, false_positives,
↪ true_negatives, false_negatives, FAR, FRR, image, {True: genuine, False: imposters
↪ }]
23         else:
24             self.rates[subject] = [true_positives, false_positives, true_negatives,
↪ false_negatives, FAR, FRR, image, {True: genuine, False: imposters}]

```

**Listing E.3:** Method for counting true positives, false positives, true negatives and false negatives.

```

1 def get_accepted_rejected(self, image, threshold):
2     """
3     Take an image filename as input and return a tally of
4     accepted and rejected images, as well as image filenames
5     """
6
7     accepted = [0, 0] # [true_positives, false_positives]
8     rejected = [0, 0] # [true_negatives, false_negatives]
9     genuine = [] # True positive image files
10    imposters = [] # False positive image files
11    for img, dist in self.hamming_distances[image].items():
12        if dist[1 + self.output_format] < threshold:
13            accepted[not dist[0]] += 1
14            if dist[0]:
15                genuine.append(img)
16            else:
17                imposters.append(img)
18        else:
19            rejected[dist[0]] += 1
20
21    return accepted, rejected, genuine, imposters

```

**Listing E.4:** Method for calculating average FAR and FRR for current threshold.

```
1 def calculate_avg_error_rate(self):
2     """Calculate and return the average error rate"""
3
4     FAR_sum = 0
5     FRR_sum = 0
6     for k, v in self.rates.items():
7         FAR_sum += v[4]
8         FRR_sum += v[5]
9
10    try:
11        FAR = FAR_sum / len(self.rates)
12        FRR = FRR_sum / len(self.rates)
13    except ZeroDivisionError:
14        FAR = 0
15        FRR = 1
16
17    return FAR, FRR
```

**Listing E.5:** Method for plotting average FAR and FRR for all threshold values.

```
1 def plot_avg_error(self, FAR, FRR):
2     """Plot graph of average error with varying thresholds"""
3
4     x_axis = [i for i in range(self.get_hash_length()+1)]
5     plt.plot(x_axis, FAR, color="blue", label="FAR")
6     plt.plot(x_axis, FRR, color="red", label="FRR")
7     plt.xlabel("Threshold")
8     plt.ylabel("Error")
9     plt.grid(color="gray", linestyle="--", linewidth=0.5)
10    plt.legend()
11    plt.show()
```



# F | Few\_pixels Attack Script

Listing F.1 shows the implementation of the *adv2\_few\_pixels\_attack.py* script used for this thesis.

Listing F.1: Modified *adv2\_few\_pixels\_attack.py* script.

```
1 import argparse
2 import math
3 import os
4 from os.path import isfile, join
5 from random import randint
6
7 import numpy as np
8 import torch
9 import Levenshtein
10 from onnx import load_model
11 from torchvision.transforms.functional import resize
12 from tqdm import tqdm
13
14 from models.neuralhash import NeuralHash
15 from losses.mse_loss import mse_loss
16 from losses.quality_losses import ssim_loss
17 from utils.hashing import compute_hash, load_hash_matrix
18 from utils.image_processing import load_and_preprocess_img, save_images
19 from utils.logger import Logger
20
21
22 def main():
23     # Parse command-line arguments
24     parser = argparse.ArgumentParser(
25         description='Perform neural collision attack.')
26     parser.add_argument('--source', dest='source', type=str,
27                         default='inputs/source.png', help='image to manipulate')
28     parser.add_argument('--learning_rate', dest='learning_rate', default=1.0,
29                         type=float, help='step size of PGD optimization step')
30     parser.add_argument('--optimizer', dest='optimizer', default='Adam',
31                         type=str, help='kind of optimizer')
32     parser.add_argument('--steps', dest='steps', default=15,
33                         type=int, help='number of optimization steps per setting')
34     parser.add_argument('--max_pixels', dest='max_pixels', default=150,
```

```

35         type=int, help='maximal number of pixels to modify')
36     parser.add_argument('--hamming', dest='hamming', default=0.01, type=float,
37                         help='proportion of bits changed in the NeuralHash')
38     parser.add_argument('--optimize_resized', dest='opt_resized',
39                         default=True, type=bool, help='optimize the resized 360x360 image
↳ ')
40     parser.add_argument('--ssim_weight', dest='ssim_weight', default=0,
41                         type=float, help='weight of ssim loss')
42     parser.add_argument('--experiment_name', dest='experiment_name',
43                         default='change_hash_attack_few_pixels', type=str, help='name of
↳ the experiment and logging file')
44     parser.add_argument('--output_folder', dest='output_folder',
45                         default='few_pixels_attack_outputs', type=str, help='folder to
↳ save optimized images in')
46     parser.add_argument('--sample_limit', dest='sample_limit',
47                         default=1000000, type=int, help='Maximum of images to be
↳ processed')
48     args = parser.parse_args()
49
50     # Create temp folder
51     os.makedirs('./temp', exist_ok=True)
52
53     # Load model and source image
54     device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
55     seed = load_hash_matrix()
56     seed = torch.tensor(seed).to(device)
57     id = randint(1, 100000)
58     temp_img = f'curr_image_{id}'
59     model = NeuralHash()
60     model.load_state_dict(torch.load('./models/model.pth'))
61     model.to(device)
62
63     # Prepare output folder
64     try:
65         os.mkdir(args.output_folder)
66     except:
67         if not os.listdir(args.output_folder):
68             print(f'Folder {args.output_folder} already exists and is empty.')
69         else:
70             print(
71                 f'Folder {args.output_folder} already exists and is not empty.')
72
73     # Prepare logging
74     logging_header = ['file', 'optimized_file', 'l2',
75                     'l_inf', 'ssim', 'steps', 'target_loss', 'num_pixels']
76     logger = Logger(args.experiment_name, logging_header, output_dir='./logs')

```

```

77 logger.add_line(['Hyperparameter', args.source, args.learning_rate,
78                 args.optimizer, args.ssim_weight, args.steps, args.max_pixels])
79 model.to(device)
80
81 # Load images
82 if os.path.isfile(args.source):
83     images = [args.source]
84 elif os.path.isdir(args.source):
85     images = [join(args.source, f) for f in os.listdir(
86               args.source) if isfile(join(args.source, f))]
87     images = sorted(images)
88 else:
89     raise RuntimeError(f'{args.source} is neither a file nor a directory.')
90 images = images[:args.sample_limit]
91
92 # define loss function
93 loss_function = mse_loss
94
95 # Start optimizing images
96 for img in tqdm(images):
97     # Store and reload source image to avoid image changes due to different formats
98     source = load_and_preprocess_img(img, device)
99     input_file_name = img.rsplit(sep='/', maxsplit=1)[1].split('.')[0]
100    save_images(source, args.output_folder, f'{input_file_name}')
101    source = load_and_preprocess_img(
102        f'{args.output_folder}/{input_file_name}.png', device)
103    orig_image = source.clone()
104
105    # Compute original hash
106    with torch.no_grad():
107        outputs_unmodified = model(source)
108        unmodified_hash_bin = compute_hash(
109            outputs_unmodified, seed, binary=True)
110        unmodified_hash_bin_str = compute_hash(
111            outputs_unmodified, seed, binary=True,
112            as_string=True)
113        unmodified_hash_hex = compute_hash(
114            outputs_unmodified, seed, binary=False)
115
116    # Compute set of pixel locations and gradient mask
117    pixel_locations = set()
118    grad_mask = torch.zeros_like(source)
119    for i in range(args.max_pixels):
120
121        # Set up optimizer
122        source.requires_grad = True

```

```

123     if args.optimizer == 'Adam':
124         optimizer = torch.optim.Adam(
125             params=[source], lr=args.learning_rate)
126     elif args.optimizer == 'SGD':
127         optimizer = torch.optim.SGD(
128             params=[source], lr=args.learning_rate)
129     else:
130         raise RuntimeError(
131             f'{args.optimizer} is no valid optimizer class. Please select --
↪ optimizer out of [Adam, SGD]')
132
133     step_size_up = math.floor(args.steps/2)
134     step_size_down = math.ceil(args.steps/2)
135
136     scheduler = torch.optim.lr_scheduler.StepLR(
137         optimizer, step_size=5, gamma=0.5)
138
139     # Compute pixels with the largest gradient (l1 norm)
140     if args.opt_resized:
141         outputs_source = model(source)
142     else:
143         outputs_source = model(resize(source, (360, 360)))
144     target_loss = - \
145         loss_function(outputs_source, unmodified_hash_bin, seed, c=1)
146     total_loss = target_loss
147     total_loss.backward()
148     grad = source.grad
149
150     # Identify pixels with largest gradient norm
151     grad_norm = torch.norm(grad, p=1, dim=1)
152     indices = grad_norm.flatten().topk(k=args.max_pixels)[1]
153     topk_indices = np.array(np.unravel_index(
154         indices.cpu().numpy(), grad_norm.shape)).T[:, 1:]
155     for k in range(args.max_pixels):
156         pixel_tuple = tuple(topk_indices[k])
157         if pixel_tuple not in pixel_locations:
158             pixel_locations.add(pixel_tuple)
159         break
160
161     # Compute new grad mask
162     for x, y in pixel_locations:
163         grad_mask[:, :, x, y] = 1
164
165     print(f'optimizing {int(grad_mask.sum().cpu().numpy()/3)} pixels')
166     for j in range(args.steps):
167         with torch.no_grad():

```

```

168         source.data = torch.clamp(source.data, min=-1, max=1)
169         source.requires_grad = True
170         if args.opt_resized:
171             outputs_source = model(source)
172         else:
173             outputs_source = model(resize(source, (360, 360)))
174         target_loss = -mse_loss(outputs_source,
175                                 unmodified_hash_bin, seed)
176         visual_loss = -ssim_loss(orig_image, source)
177         optimizer.zero_grad()
178         total_loss = target_loss + args.ssim_weight * visual_loss
179         total_loss.backward()
180         optimizer.param_groups[0]['params'][0].grad *= grad_mask
181         optimizer.step()
182         scheduler.step()
183         print(
184             f'Iteration {j+1}: \tTarget Loss {target_loss.detach():.4f}, Visual
↪ Loss {visual_loss.detach():.4f}')
185
186         # Check for hash changes after optimizing pixels
187         with torch.no_grad():
188             save_images(source, './temp', temp_img)
189             current_img = load_and_preprocess_img(
190                 f'./temp/{temp_img}.png', device)
191             check_output = model(current_img)
192             source_hash_hex = compute_hash(check_output, seed, binary=False)
193             source_hash_bin = compute_hash(check_output, seed, binary=True, as_string
↪ =True)
194             print(
195                 f'Optimizing {i+1} pixels: Original Hash: {unmodified_hash_hex},
↪ Current Hash: {source_hash_hex}')
196             # Log results and finish if hash has changed sufficiently
197             if Levenshtein.hamming(source_hash_bin, unmodified_hash_bin_str) / 96 >
↪ args.hamming:
198                 optimized_file = f'{args.output_folder}/{input_file_name}_opt'
199                 save_images(source, args.output_folder,
200                             f'{input_file_name}_opt')
201                 # Compute metrics in the [0, 1] space
202                 l2_distance = torch.norm(
203                     ((current_img + 1) / 2) - ((orig_image + 1) / 2), p=2)
204                 linf_distance = torch.norm(
205                     ((current_img + 1) / 2) - ((orig_image + 1) / 2), p=float("inf"))
206                 ssim_distance = ssim_loss(
207                     (current_img + 1) / 2, (orig_image + 1) / 2)
208                 print(
209                     f'Finishing after {i+1} steps - L2 distance: {l2_distance:.4f} -

```

```

↪ L-Inf distance: {linf_distance:.4f} - SSIM: {ssim_distance:.4f}')
210
211         logger_data = [img, optimized_file + '.png', l2_distance.item(),
212                        linf_distance.item(), ssim_distance.item(), (i*args.
↪ max_pixels) + j+1, target_loss.item(), i+1]
213         logger.add_line(logger_data)
214         break
215         # if source_hash_hex != unmodified_hash_hex:
216         if Levenshtein.hamming(source_hash_bin, unmodified_hash_bin_str) / 96 > args.
↪ hamming:
217         print(
218             f'Finishing optimization after {i+1} iterations and {int(grad_mask.
↪ sum().cpu().numpy()/3)} optimized pixels.')
219         break
220     logger.finish_logging()
221
222
223 if __name__ == "__main__":
224     main()

```