CHALMERS
UNIVERSITY OF TECHNOLOGY

# Automatic Testing of Graphical User Interfaces

A comparison of using edge detection and neural networks to identify interactive areas in graphical user interfaces.

Master's thesis in Systems, Control and Mechatronics

Heinerud, Joel
Nilsson, Tomas

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis EX057/2017

# Automatic Testing of Graphical User Interfaces

A comparison of using edge detection and neural networks to identify interactive areas in graphical user interfaces.

Joel Heinerud
Tomas Nilsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
*Division of Systems and Control*
Chalmers University of Technology
Gothenburg, Sweden 2017

Automatic Testing of Graphical User Interfaces
A comparison of using edge detection and neural networks to identify interactive areas in graphical user interfaces.
Joel Heinerud Tomas Nilsson

© Joel Heinerud, 2017.
© Tomas Nilsson, 2017.

Supervisor: Mikael Lundgren, Benchnode Technology AB
Examiner: Fredrik Kahl, Deptartment of Electrical Engineering, Chalmers University of Technology

Master's Thesis EX057/2017
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Automatic testing of graphical user interfaces
A comparison of using edge detection and neural networks to identify interactive areas in graphical user interfaces
Joel Heinerud, Tomas Nilsson
Department of Signals and Systems (S2)
Chalmers University of Technology

## Abstract

Automatic testing is a part of the development process when creating GUIs. Traditionally, this is a time consuming process that is performed both by the supplier, and the company ordering the GUI. By using image processing, this process can be performed dynamically without having to specify rules manually. In this thesis, two methods for identifying interactive areas have been studied. These methods are identification using edge detection as well as convolutional neural networks. The evaluation was done by comparing how accurate these methods were when faced with GUIs of varying complexity. The neural network approach was found to perform better in all aspects, which led to a testing framework that can navigate through a novel GUI automatically. The result of searching through a GUI is presented in the form of a graph which can be used to study GUI complexity.

Keywords: Touch-Screen, GUI, Image Analysis, Computer Vision, Automation, Test Evaluation, Machine Learning, Deep Learning, Perceptual Hashing, GoogLeNet, Inception module.

# Acknowledgements

This thesis was produced at Benchnode Technology AB, located at Lindholmen, Gothenburg. We would like to express out deepest gratitude to them for letting us conduct our work using the office of Benchnode as well as their equipment. The support from Mikael Lundgren, Mattias Fredriksson, Mikael Westerlind and other employees who have given us valuable insights and tips along the way has been very helpful. We would like to thank Fredrik Kahl, our examiner for his insights and help with the report. Additional thanks to the wonderful community of the TensorBox Gitter page, who have shared their tips and tricks, as well as tales of troubles and tribulations so that others may be spared from making the same mistakes. Finally, thanks to our friends and families for your support.

Joel Heinerud and Tomas Nilsson, Gothenburg, May 2017

# Contents

**Abbreviations** **xi**

**Nomenclature** **xi**

**List of Figures** **xiii**

**List of Tables** **xv**

**1 Introduction** **1**
1.1 Related Work . . . . . 2
1.2 Purpose . . . . . 2
1.3 Contributions . . . . . 3

**2 Preliminaries** **5**
2.1 Problem Description . . . . . 5
2.2 General Objectives . . . . . 5
2.3 Limitations . . . . . 6

**3 Theory** **9**
3.1 Perceptual Hashing . . . . . 9
3.1.1 Average Based Hash (aHash) . . . . . 10
3.1.2 Block Based Hash (bHash) . . . . . 10
3.1.3 Gradient Based Hash (dHash) . . . . . 11
3.1.4 DCT Based Hash (pHash) . . . . . 11
3.1.5 Hamming Distance . . . . . 12
3.2 Edge Detection . . . . . 12
3.2.1 Gaussian Filtering . . . . . 12
3.2.2 Sobel Filtering . . . . . 14
3.3 Neural Networks . . . . . 14
3.3.1 The Makings of a Neural Network . . . . . 15
3.3.1.1 Convolutional Layer . . . . . 16
3.3.1.2 Pooling Layer . . . . . 18
3.3.1.3 Concatenation Layer . . . . . 18
3.3.1.4 Dropout Layer . . . . . 19
3.3.1.5 Fully Connected Layer . . . . . 20
3.3.1.6 Softmax Layer . . . . . 20
3.3.1.7 Activation Function . . . . . 20

3.3.2 Training a Neural Network . . . . . . . . . . . . . . . . . . . . 21
3.3.2.1 Training Set . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.2.2 Validation Set . . . . . . . . . . . . . . . . . . . . . . 22
3.3.2.3 Testing Set . . . . . . . . . . . . . . . . . . . . . . . 23
3.3.2.4 Overfitting . . . . . . . . . . . . . . . . . . . . . . . 23
3.3.3 GoogLeNet . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3.3.1 The Inception Module . . . . . . . . . . . . . . . . . . 24

**4 Method 27**
4.1 Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.1 MATLAB . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.2 Android Device . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.3 Android Studio . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.3.1 Android Debug Bridge . . . . . . . . . . . . . . . . . . 28
4.1.4 Python . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.5 TensorFlow . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.5.1 TensorBoard . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.5.2 TensorBox . . . . . . . . . . . . . . . . . . . . . . . . 29
4.2 Data Acquisition . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3 Identification . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3.1 Edge Detection Method . . . . . . . . . . . . . . . . . . . 30
4.3.2 Identification Using TensorBox . . . . . . . . . . . . . . . 32
4.3.2.1 Preparing Training Data . . . . . . . . . . . . . . . . . 32
4.4 Classification . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.5 System Modelling . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.6 GUI Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.7 Proposed Method . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.7.1 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 35

**5 Results 37**
5.1 Identification . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.2 Classification . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.3 Test Runs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**6 Discussion and Concluding Remarks 43**
6.1 Further Development . . . . . . . . . . . . . . . . . . . . . . . . 44

**A Appendix 1 I**
A.1 Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . I
A.2 Hashing comparisons . . . . . . . . . . . . . . . . . . . . . . . . V

**B Appendix 2 VII**
B.1 Python Packages . . . . . . . . . . . . . . . . . . . . . . . . . . VII
B.2 Android Debug Bridge: Code samples . . . . . . . . . . . . . . . VIII

**Bibliography IX**

# Abbreviations

| | |
|---|---|
| adb | Android Debug Bridge |
| CNN | Convolutional Neural Network |
| CONV | Convolutional layer |
| DCT | Discrete Cosine Transform |
| GUI | Graphical User Interface |
| JSON | JavaScript Object Notation |
| MSE | Mean Squared Error |
| NN | Neural Network |
| ReLu | Rectified Linear Units |

# Nomenclature

| | |
|---|---|
| $d_h$ | Hamming Distance |
| $\eta$ | Learning rate |
| $f_k$ | Kernel size |
| $f_s$ | Stride length during a filter operation |
| $I_N$ | A normalized image |
| $P$ | Padding size to add to an input volume |
| $r$ | Result of comparision |
| $u_0$ | The initial image from the GUI |
| $u_{ss}$ | Steady state image of GUI |
| $x$ | Identified areas of interest |
| $\hat{x}$ | Classified objects |
| $\hat{x_i}$ | Selected object to try an interaction on |
| $z_I$ | Input volume size of a NN-layer |
| $z_O$ | Output volume size of a NN-layer |

# List of Figures


3.1 Visualization of the Gaussian filter . . . . . 13
3.2 Neural network basics . . . . . 15
3.3 Zeiler-Fergus Architecture . . . . . 16
3.4 Convolutional layer visualization . . . . . 16
3.5 Image Convolution . . . . . 17
3.6 Stride lengths effect on output size . . . . . 17
3.7 Maximum pool visualization . . . . . 18
3.8 Concatenation Layer . . . . . 19
3.9 Visualisation of dropouts effect on a neural network[1]. . . . . . 19
3.10 Learning rate . . . . . 22
3.11 Overfitting . . . . . 23
3.12 GoogLeNet Inception v.3 . . . . . 24
3.13 Inception module . . . . . 24
3.14 Inception module with dimension reductions[2]. . . . . . 25

4.1 Schematic view of the early system. . . . . . 27
4.2 Grayscale conversion . . . . . 30
4.3 Results of Gaussian filtering . . . . . 31
4.4 Sobel filtering . . . . . 31
4.5 Boundary boxes . . . . . 32
4.6 Training image before and after resizing. . . . . . 33
4.7 The three steps of the annotation process. . . . . . 33
4.8 The proposed workflow. . . . . . 36

5.1 Identified icons on an iPhone GUI with a plain background. . . . . . 38
5.2 Identified icons on an iPhone GUI with a background that has a color gradient. . . . . . 38
5.3 Identified icons on a generic image with a slight gradient. . . . . . 39
5.4 Example output, simple . . . . . 41
5.5 Example output, complicated . . . . . 42

A.1 The proposed algorithm . . . . . II
A.2 State diagram and transitions over the used program . . . . . III
A.3 Resulting state machine of a early terminated run on a complex GUI structure. . . . . . IV
A.4 Hashing test images . . . . . V
A.5 Hashing test images . . . . . VI

# List of Tables


5.1 Identification results, plain background. . . . . . . . . . . . . . . . . . . . 37
5.2 Hamming distance of the dHash values between the images in Figure A.4, downscaled to 7x6 . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.3 Hamming distance between the dHash-values of the images in Figure A.4, downscaled to 11x10 . . . . . . . . . . . . . . . . . . . . . . . . 40

# 1 Introduction

The use of touch screens have become increasingly widespread since they have proven to excel in multiple applications over the last few years. The expansive smartphone and tablet markets have helped speed up the transition from traditional, physical knobs and switches to virtual, graphical user interfaces (GUI). In the automotive industry, this transition has enabled greater flexibility when designing the interior of the car. It also enables for the possibility to update and change the look and functionality of the dashboard even after the car has been manufactured simply by updating the software.

Part of the late stage testing of software, either completely new software, or just a new version, is testing of the GUI. The GUI can vary greatly from version to version, and sometimes even within versions depending on user settings. It is necessary to verify that old functions are not broken by each new release. This is called regression testing. Regression testing is usually a trivial task for a human, it can however be rather time consuming. To an automated testing system, this might not be the case and is often a quite complex problem. Fonts, window placement, colors, graphics, sizes and a number of other things may have been altered in the software update, complicating testing between versions.

Currently there exist some automated GUI testing solutions, but they require precise and time consuming configurations that can be complex, and they will need to be changed every time an object in the GUI is changed, added or removed. Most existing solutions, such as the "Linux desktop testing project"[3], aim to test a predefined scenario and are not able to test unknown GUIs. Programs like this often require access to the event stream of the software to capture and send events to traverse the program. These system tests are generally performed manually to find pieces of the software that is not working as intended.

Benchnode Technology AB is currently supplying NEVS with various test strategies for verifying the integrity and function of the CAN network. A growing need to test the touchscreen in the vehicle in an orderly manner has been identified. As such, an efficient, automated, and general testing solution for late stage testing of an unknown touch screen GUI is desired.

## 1.1 Related Work

Evaluating a GUI using image analysis techniques has been suggested as a potential improvement in existing testing solutions. The article *Automatic GUI Test by using SIFT matching [4]* compares using methods such as SIFT versus Random FERN for identifying features in an image. The article suggest a combination between the two for identification of image features.

Once objects have been identified, a method of conducting the tests is needed. Monkey testing[5] is, as the name suggests, a rather naive approach to testing. The system will try to find flaws in the GUI by applying more or less random inputs. A benefit of monkey testing is that little or no information about the system is needed, which makes it useful when testing unknown systems. A downside is that due to the randomness of the test procedure, found bugs can be hard to reproduce. If a specific function of the GUI should be tested repeatedly and with speed, a systematic approach is needed. Information about how the GUI might look and behave is needed to test a given function. T. Daboczi et.al[6] proposed several approaches to GUI testing. One of which uses a state machine to keep track of the behaviour of the GUI. The testing is then carried out by using a planning algorithm based on the desired result. The accuracy of the actual result is then verified.

## 1.2 Purpose

This thesis aim to explore the possibility of using image analysis and machine learning techniques to create an adaptive and flexible testing tool for touch screen user interfaces. The solution needs to be dynamic, and suitable to many different test cases. This would reduce the need for specialized test rigs and brand new solutions for each new iteration of a product.

The main focus of the thesis will revolve around evaluating ways to determine the current interactive objects on the screen. This can then be used to interpret the current state of the system. An object is defined as an interactive (something happens if it is clicked) area of pixels within the GUI.

The thesis will aim to answer the following questions:

- What is an effective way of identifying objects in a GUI?
- What is an effective way to distinguish between interactive objects and passive areas?
- What is a suitable way to model the structure of a GUI?

A measurable goal that all other tasks revolve around, is to automatically find ob-

jects in an arbitrary touch screen GUI and test if they respond in an appropriate manner when clicked on. The algorithms behind identifying various interactive objects on the screen is the main focus of the thesis. The efficiency and accuracy of performing the task of automatically testing a user interface will lead to an evaluation process to find the best method.

The task of deciding whether the response of a certain interactive object was appropriate or not will be fulfilled if there was any visual response at all.

## 1.3 Contributions

With this thesis, it has been proven that it is possible to identify interactive objects and navigate through a novel GUI using a combination of convolutional neural network and perceptual hashing. Compared to a classical edge detection algorithm, the neural network performs with higher accuracy and above all, greater flexibility. The network has not been trained on the GUI which it is tested on, but on training data gathered from other sources. The neural network was trained on generic images of mobile phone GUIs, which proves that the GUI on which the method would be applied can indeed be unknown.

The results from building a graph structure could prove to be helpful when verifying the complexity of a GUI. When designing a car, it is important to provide the driver with simple and logical input controls. After all, the driver should be looking as little as possible at the touch screen, and instead be focused on the road and the surrounding traffic. This means that GUIs should be as shallow as possible, i.e. it should be possible to navigate to all important functions relatively quickly. However, when mapping the depth of a GUI, the identification process cannot be guaranteed to perform perfectly every time. This means that the results from mapping a GUI should be used with care.

# 2

# Preliminaries

This chapter of the thesis describes the prerequisites, defines the problem, goes through assumptions and limitations that have been made and briefly describes the set up necessary to run the proposed testing solution.

## 2.1 Problem Description

The task of testing a touch-screen GUI can mean a great number of things. In the scope of this thesis, it is defined as the task of systematically finding and testing each and every clickable area. When clicked, the GUI should react in some (preferably meaningful) way. If that is the case, the transition should be mapped into a graphical representation of the GUI. In the scope of this thesis, any kind of response will count as meaningful.

## 2.2 General Objectives

In order to overcome the problem of finding GUI objects and testing how they affect the system, several subproblems will have to be solved. These problems are in turn broken down into smaller, more manageable parts.

**Identification** - Finding possible clickable areas. Ultimately, this translates to a number of (x,y)-coordinates.

**Data acquisition** - The task of collecting data about the current state of the GUI. The data consists of images that will be used in multiple stages of the project.

**Testing** - The feat of automatically making the touch screen react to an input.

**Control** - In order to test the GUI, a way to interacting with the GUI is needed.

**Modelling** - To enable efficient search procedures and to help understand how the system is designed.

**Evaluation** - Collecting data from the previous steps and presenting it in a meaningful way to the user. This is also needed to answer the questions proposed in the preliminary aim.

These subproblems and how they were approached are described in full detail in Chapter 4.

## 2.3 Limitations

These limitations mainly focus on simplifying the problem into separate, controllable elements, and to reduce complexity.

**Fonts** can vary greatly in both style and size, adding a complexity to the identification methods needed without adding a large value to the problem. Because of this, only a single, or a small set of fonts are considered.

**Buttons** is also something that can vary a lot, from a small simple gray button to an animated graphic or icon. The button could be tied to a menu, list or another multistate function requiring multiple navigational steps to access. The buttons were assumed to be relatively symmetrical with a sharp edge and a contrasting color.

**Size** differences of the interactive objects were assumed to be limited. Tiny objects (only a few pixels wide) are unlikely to appear in a real system.

**Menu depth** is assumed to be small, since deeper menus tied behind previous menus quickly results in a combinatorial explosion of test cases. If handling a menu with two layers is successful, the solution should be expandable to menus with greater depths.

**Image quality** , such as the amount of noise, highly impacts the success rates of image analysis techniques and can introduce an element of randomness that should be avoided during the development of the algorithms intended for this thesis.

**Image resolution** can greatly affect computational complexity, thus a limit to the image resolution and size were established.

**Input methods** for a touchscreen device can vary from a simple single tap with one finger, to more complex multi-touch gestures, all with different effects depending on the state of the device. Thus, a limitation to the number of gestures were established. Only the simplest of inputs in the form of a one finger tap is considered.

**Device platform** is limited to Android due to the wide array of open source tools available for developers. This made it possible to quickly establish a commu-

nication channel between an Android device and a PC.

**GUI complexity** is limited. A simple GUI with only a few object types is considered.

**Interactive area** is limited. Parts of the screen area are limited to prevent the phone from trying to make calls, or open other unwanted applications.

# 3

# Theory

To understand how the GUI testing tool works, explanations of the theory behind the methods used are provided in this chapter. This chapter consists of brief overviews of the key theory components of the solution. The chapter describes perceptual hashing, edge detection and neural networks. More in depth explanations, especially on the matter of Convolutional Neural Networks, can be found in the references.

## 3.1 Perceptual Hashing

Being able to identify and quantify similarities between, or even in, images is an important part of the image analysis toolbox. There are a multitude of different ways to measure image similarity. Examples of such methods are; keypoint matching, where SIFT[7] is arguably the most commonly used, different image histogram[8] methods, keypoint recognition using random fern[9], or perceptual hashing[10]. This project has used perceptual hashing because it is simple to implement while providing a solution that can calculate a similarity measurement with enough speed and accuracy for the scope of this thesis.

Perceptual hashing is a way to identify and compare various forms of multimedia content, such as images, music or text. It is done by creating fingerprints of the media that are analogous if the original features are similar. This is commonly used to find content that are similar, or that have been modified slightly. Most perceptual hashing algorithms work by trying to look at the general structure of the content, instead of the details to find matches that are similar. If the hashes are stored in a database, it is a one time calculation per image to compute the perceptual hash of an image and then the comparison search for a matching hash is relatively quick.

Perceptual hashing differ from encryption hashing where a small adjustment to the data will result in a completely different hash value in the way that small adjustments to the data will only result in a small difference to the hash value, so it can be used as a comparison measurement.

### 3.1.1 Average Based Hash (aHash)

The average hash algorithm is one of the more basic perceptual hashing algorithms. It uses averages of the low frequency components of an image. The resulting hash is static even if the input image is scaled or if the aspect ratio changes. Altering brightness, contrast or colors will not change the hash value dramatically, however minor modifications such as changing text, moved objects can be missed. The basic algorithm is described below, simplified from a blog post on Dr. Neal Krawetz blog called "Hacker Factor"[11]. The average hash is very quick, but compared to other perceptual hashing methods, it is rather inaccurate since it easily misses minor modifications to an image. The algorithm operates according to:

1. Reduce color by converting the image, $I$, to grayscale
2. Reduce and normalize the image to a predetermined size
3. Compute the mean value, $M$, of the image
4. Compute the hash values by comparing each element in the image to the mean according to

$$h(i) = \begin{cases} 0, & I(i) < M \\ 1, & I(i) \geq M \end{cases}.$$

### 3.1.2 Block Based Hash (bHash)

The basic block hash function is generally performed as described by the first algorithm proposed by Bian Yang[12] et al. and is based on the mean values of blocks of the image. Block hash methods can incorporate rotational operations to better handle cases where the images being compared are rotated, or overlap parts of the selected blocks. This has been left out in the algorithm below and are described in more detail in the previously mentioned article. The block hashing algorithm adds quite a few computations. Thus, it is slower than the average hash, but in turn it is more accurate.

1. Reduce color by converting the image, $I$, to grayscale
2. Reduce and normalize the original image into a predetermined size
3. Divide the normalized image, $I_N$, into a sequence of blocks as

$$I_a = \{I(1), I(2), \ldots, I(n)\}.$$

4. Encrypt the indices of the previous sequence to obtain a new order of the blocks, $I_b$.

5. Compute the mean value sequences of the image blocks as

$$M_{a,b}(m) = \text{median}(I_{a,b}(m)) \qquad (m = 1, 2, \ldots, n).$$

6. Normalize the mean value sequence into a binary form and obtain the hash values as

$$h(i) = \begin{cases} 0, & M_a(i) < M_b(i) \\ 1, & M_a(i) \geq M_b(i) \end{cases} \qquad (i = 1, 2, \ldots, m).$$

### 3.1.3 Gradient Based Hash (dHash)

The dHash algorithm works by tracking the gradient of the images[13]. It utilizes very few operations, is very fast and it is simple to implement. As with the aHash algorithm, dHash is not affected by scaling or aspect ratio operations. The algorithm also robustly handles other image operations such as gamma correction and color profile changes. Changes in brightness or contrast will not significantly change the resulting hash either. The simplicity in the dHash algorithm makes it very fast to generate the hashes, about equally fast as the average hashing algorithm. Its accuracy is better than the block hash algorithm[14], and nearly as good as the DCT based hash [13], which is covered in the next section.

1. Reduce color by converting the image, $I$, to grayscale
2. Reduce and normalize the original image into a preset size, $I$
3. Compute the relative gradient direction and obtain the hash values as

$$h(i) = \begin{cases} 0, & I(i) < I(i+1) \\ 1, & I(i) \geq I(i+1) \end{cases}.$$

### 3.1.4 DCT Based Hash (pHash)

The pHash open source perceptual hash library[15] implements an algorithm that creates the hash by evaluating color frequency patterns in the images using a Discrete Cosine Transform[16]. When computing the mean of the DCT, the first term is removed, which excludes flat, solid color and image information from the hash. The computational time of a DCT based hash is quite high, but the accuracy of the algorithm is high[17].

A simplification of the algorithm is provided below.

1. Reduce color by converting the image, $I$, to grayscale
2. Reduce and normalize the image to a preset size

3. Compute the DCT to separate the image into a collection of frequencies and scalars as $I_{dct}$
4. Reduce the DCT to just contain the lowest frequencies in the image by selecting the top-left $8 \times 8$ values of $I_{dct}$
5. Compute the mean, $M$, of $I_{dct}$, excluding the first term.
6. Reduce the DCT further by comparing each value of the $I_{dct}$ to the mean value and turn it into a binary form and obtain the hash as

$$h(i) = \begin{cases} 0, & I_{dct,i} < M \\ 1, & I_{dct,i} \geq M \end{cases}.$$

### 3.1.5 Hamming Distance

The Hamming distance ($d_h$)[18] for two strings of symbols of equal length is the number of substitutions required for one string to be equal to the other. In a set of two binary strings of length $n$ this is equal to the number of flipped bits between the two.

For example, $A = 0101$ & $B = 1010$ has a Hamming distance of $d_h = \sum_{i=0}^{n-1} |A_i - B_i| = 4$, since all bits between the two are different.

The Hamming distance can be used as a similarity measurement between two perceptual hash strings of equal length, provided the two hashes were constructed using the same algorithm. This is useful in cases where small changes between the compared objects should be allowed, if it is under a certain threshold.

## 3.2 Edge Detection

Edge detection is a set of mathematical methods that aims to identify changes in intensity (brightness) in an image. In this section, Gaussian filtering and the Sobel operator are briefly described.

### 3.2.1 Gaussian Filtering

When performing edge detection, it is often the major gradients that are the most interesting. It is usually the case that some kind of lowpass filtering has to be performed before applying the edge detection filter in order to obtain good results. Otherwise, it is hard to discriminate between the actual edges in an image, and edges appearing due to noise.

The value of each pixel in an image is recalculated using a filter according to the Gaussian function according to

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}. \tag{3.1}$$

In the two-dimensional case, seen in Figure 3.1, the Gaussian function is instead defined as

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.2}$$

where $(x, y)$ is the horizontal and vertical distance to the center of the filter for that particular coordinate and $\sigma$ is the standard deviation of the Gaussian distribution. This yields a circular filter that is converted to a $n$ by $n$ matrix which then can be convolved with the image. The size $n(\sigma)$ of the matrix calculated using

$$n(\sigma) = 2 \times ceil(2 \times \sigma) + 1 \tag{3.3}$$

which will always produce an odd filter size. The *ceil()* function rounds up to the nearest integer.

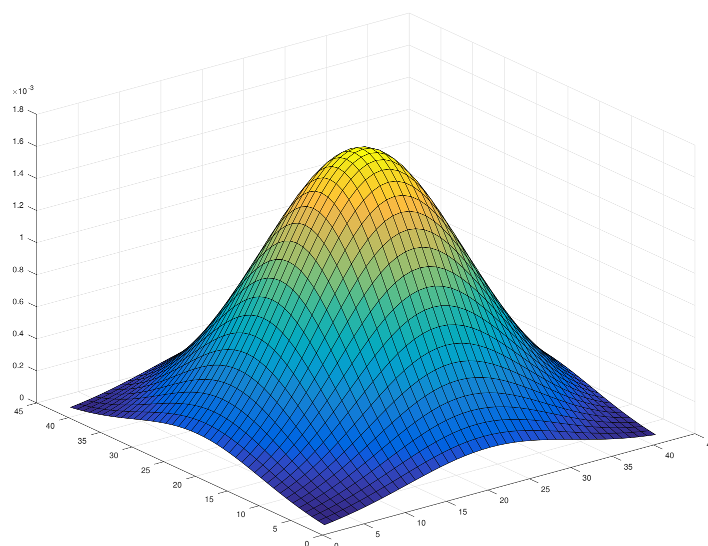



**Figure 3.1:** Visualization of the Gaussian filter with $\sigma = 10$.

A Gaussian filter with $\sigma = 1$ will produce a $5 \times 5$ matrix according to

$$\frac{1}{100}\begin{bmatrix} 0.2969 & 1.3306 & 2.1938 & 1.3306 & 0.2969 \\ 1.3306 & 5.9634 & 9.8320 & 5.9634 & 1.3306 \\ 2.1938 & 9.8320 & 16.2103 & 9.8320 & 2.1938 \\ 1.3306 & 5.9634 & 9.8320 & 5.9634 & 1.3306 \\ 0.2969 & 1.3306 & 2.1938 & 1.3306 & 0.2969 \end{bmatrix}.$$

### 3.2.2 Sobel Filtering

The Sobel operator is the filtering that performs the heavy lifting of the edge detection process. The result of a Sobel operation is an approximation of the pixel intensity derivative. It consists of two $3 \times 3$ matrices $S_x$ and $S_y$ (one for each axis), where

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad and \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.4}$$

In order to produce an approximation of the intensity derivatives, $S_x$ and $S_y$ are convolved with an image $I$ resulting in two new matrices $G_x$ and $G_y$ according to

$$G_x = S_x * I \quad and \quad G_y = S_y * I \tag{3.5}$$

Combining $G_x$ and $G_y$ as

$$G = \sqrt{G_x^2 + G_y^2} \tag{3.6}$$

gives the resulting total gradient magnitude.

## 3.3 Neural Networks

Artificial neural networks is a computational model used in the field of machine learning that is inspired by the way biological neural networks process information. Just like humans, artificial neural networks need to learn from a set of known data before they can be expected to solve a problem successfully. The goal is to infer different levels of abstract rules from these training sets, and by learning these rules, the accuracy of the neural network classifiers hopefully increase.

Neural networks and machine learning have been used to solve a wide array of different problems, not limited to image processing. Even though, at the time of writing

this thesis, classification using images is arguably the most common application of neural networks.

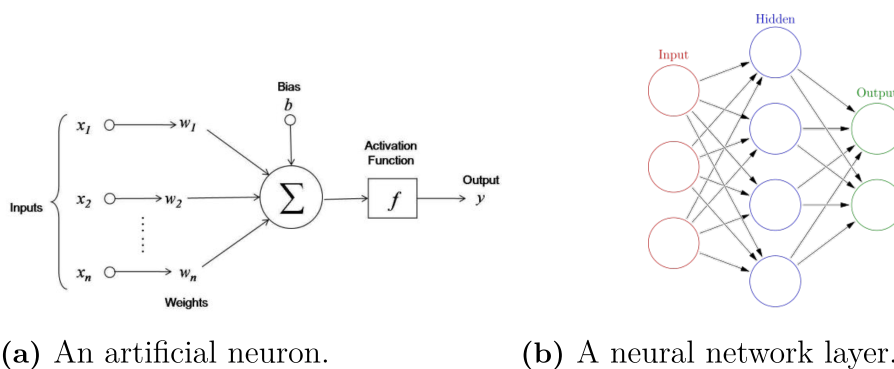
This section aims to give a brief explanation of the core concepts when using a neural network as an image classifier.

### 3.3.1 The Makings of a Neural Network

The elemental part of the neural network is the artificial neuron, Figure 3.2a. The mathematical model of the neuron, seen in Equation (3.7), is used to sum up each weighted input along with a bias, $b$. This sum is then fed through an activation function, $f$, to produce the output, $y$. The activation function is described further in Section 3.3.1.7.

$$y = f\left(\sum_{i=1}^{n}(w_i x_i) + b\right) \tag{3.7}$$

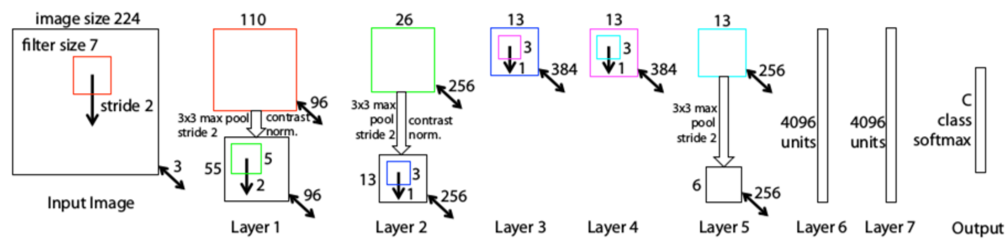
These neurons are then arranged in so called layers, a common representation of this can be seen in Figure 3.2b where each node represents an artificial neuron. The weights of the neurons in the layers are adjusted when training a network to work on a certain problem.



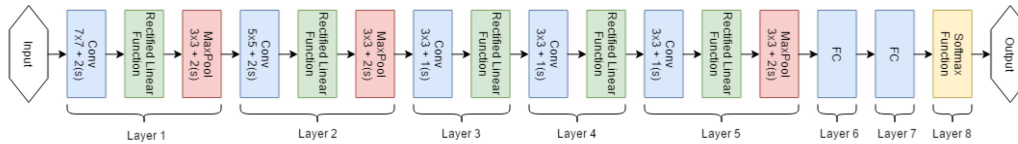

**(a)** An artificial neuron. **(b)** A neural network layer.

**Figure 3.2:** The basics of a neural network, the neuron, and a layer of neurons[19].

These layers can vary in their form and function, and different types of layers will be discussed in the coming sections. A large part of designing a network is arranging all these different layers. Traditionally the classical arrangement has been to simply put one layer after the other, such as in the 8 layer Zeiler-Fergus Architecture[20], seen in Figure 3.3. Lately, more complex architectures has emerged, such as the GoogLeNet discussed in Section 3.3.3.




**(a)** As visualised by M. Zeiler and R. Fergus[20].



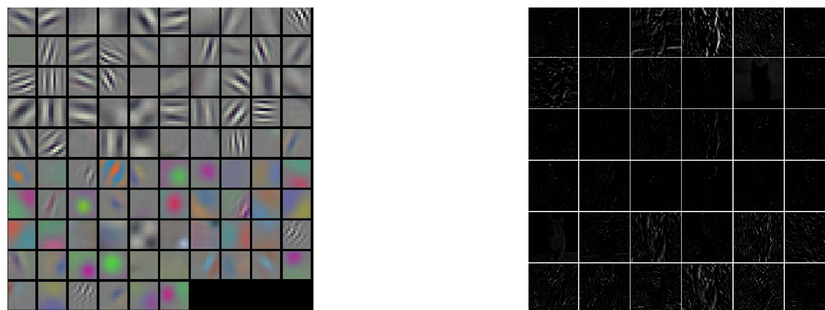

**(b)** A simplified tree structure.

**Figure 3.3:** The 8 layer Zeiler-Fergus architecture neural network.

#### 3.3.1.1 Convolutional Layer

A convolutional layer (CONV) is the main part of a convolutional neural network (CNN) and it is in essence a feature identifier. Early in the network, the features are simple characteristics such as edges, lines, curves and colors The features they describe get more abstract further in through the network.

It's first after training that the CONV weights will correspond to the different features. An example of a trained layer visualiation can be seen in Figure 3.4 where some low level features can be identified, such as straight lines, colours and the like.



**(a)** Visualization of the weights.

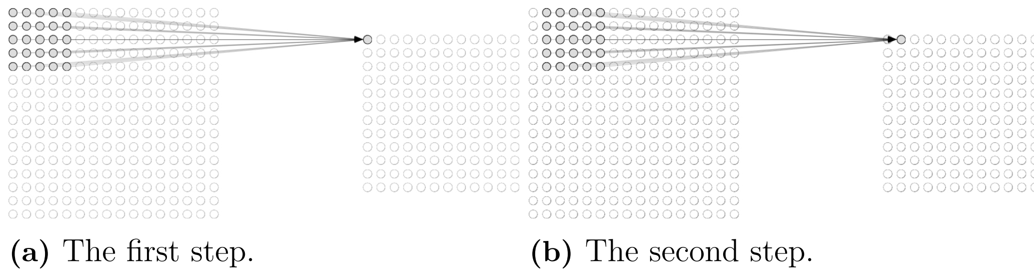
**(b)** The activation map looking at a picture of a cat.

**Figure 3.4:** Typical visualization of a trained first layer CONV[21].

The CONV performs a convolution between the input from a previous layer and a filter and outputs the result to the next layer in the network.

The convolutional layer slides a filter (a kernel) over the input. The region covered

by the kernel at a certain position on the input is called local receptive field. Each local receptive field position corresponds to a hidden neuron in an array as it slides over the input, this results in an so called activation map. It is sometimes also referred to as a feature map as it corresponds to a certain feature. An illustration of two steps of the convolution can be seen in Figure 3.5.



**(a)** The first step. **(b)** The second step.

**Figure 3.5:** The local receptive field ($5 \times 5$, $f_k = 5$) slides through the input ($15 \times 15$, $z_I = 15$) with a stride length of one and connects to a corresponding hidden neuron.

A CONV has three so called hyper parameters that define it. The parameters are, kernel size, $f_k$, stride, $f_s$, and padding, $P$. Together with the input volume size, $z_I$, they dictate the size of the output volume, $z_O$. These parameters can be tuned to achieve different results, such as a reduction in output volume to reduce complexity. In Figure 3.6 an example on the effects of the stride length on output size can be seen.

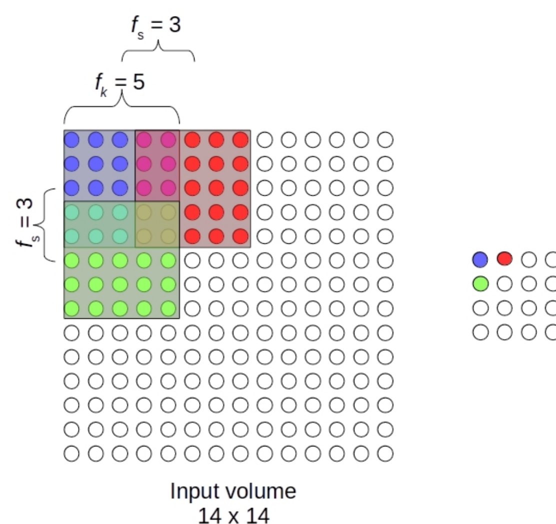



**Figure 3.6:** Visualization of the effect of stride length. Stride is here set to three, and with a kernel size of 5, the input volume ($14 \times 14$) becomes a $4 \times 4$ matrix.

#### 3.3.1.2 Pooling Layer

Once a specific feature has been identified in the previous layer, the feature's exact location is not as relevant as the relative position to other features in the input volume. To accomplish this, a pooling layer is applied, where the most commonly used is a max-pool. By applying a maximum filter, only the largest responses in each subregion is considered and sent through to the next layer. It is a sliding filter with the same stride length as the filter size, outputting the maximum value in every subregion. An example of this can be seen in Figure 3.7.

Other pooling functions are sometimes used, such as average pooling or $\ell^2$-norm pooling. In average pooling, the average of each subregion is fed forward through the layer, $\bar{x} = \frac{(x_1+...+x_n)}{n}$. Likewise $\ell^2$-norm pooling feeds the euclidean distance, $\|x\| = \sqrt{x_1^2 + ... + x_n^2}$ to the next layer.

This layer also reduces the dimension of the input, where the reduction factor depends on the filter size and stride length. This helps keeping computational cost down, as well as providing a way to reduce the risk of overfitting the network on the training set.

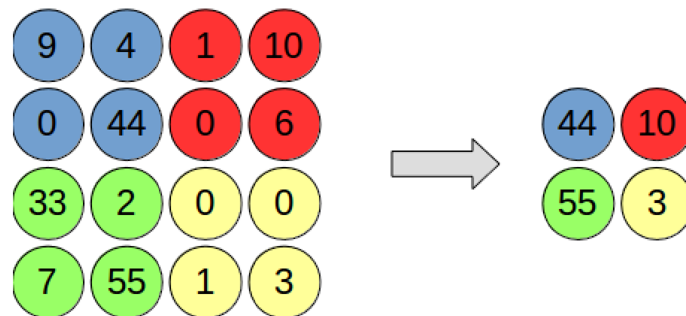



**Figure 3.7:** Visualization of a maximum pool with a filter size and stride length of two. This is reducing the input volume dimension by 75%, while keeping the features relative position.

#### 3.3.1.3 Concatenation Layer

The concatenation layer is very self descriptive, it concatenates the different filter outputs into a single output. This is for example used as the last layer in an inception module, as described in Section 3.3.3.1, to gather the output from the parallel filters into a one output, as seen in Figure 3.8.

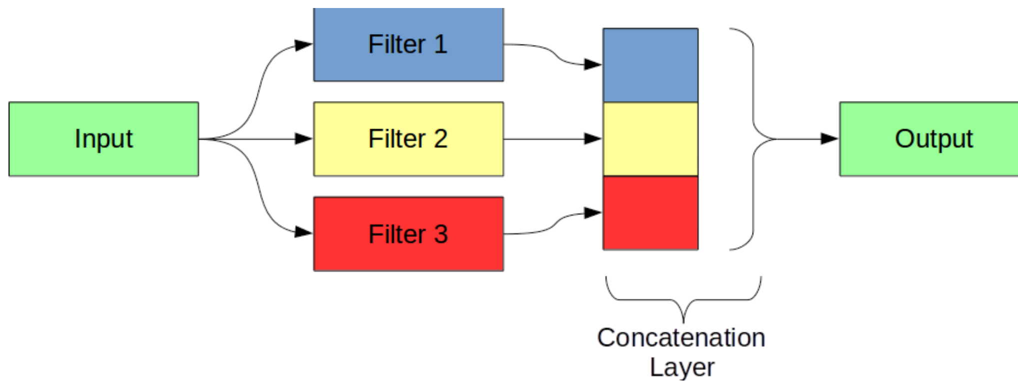
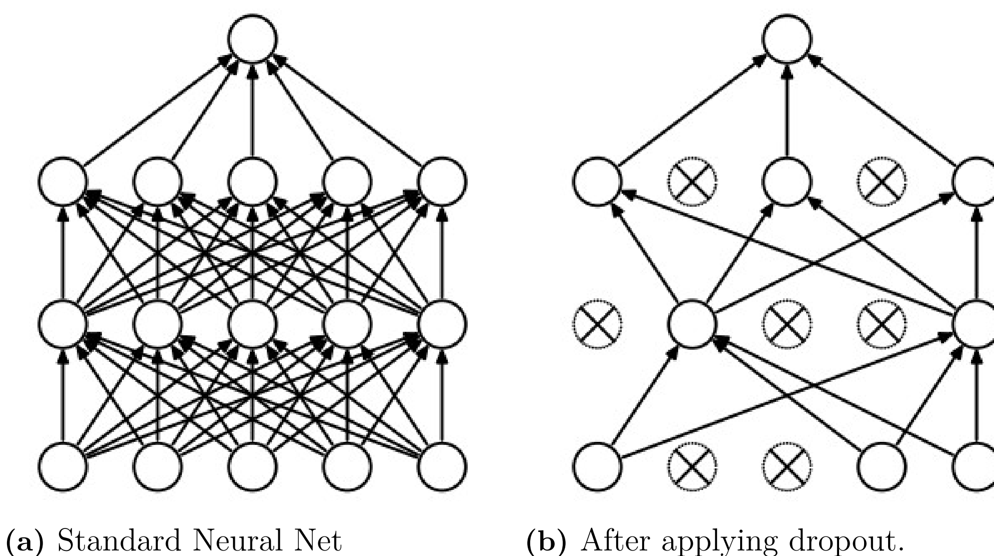



**Figure 3.8:** Visualization of a concatenation of three parallel filters into one output.

#### 3.3.1.4 Dropout Layer

The dropout layer[1] is a layer that is only used during the training of the network, not during the testing or the actual use of the resulting filter. The main purpose of the dropout layer is to help reduce overfitting, described in Section 3.3.2.4, by forcing the network to have built in redundancies. It does this simply by "dropping out" some activations. That is, it sets a random set of activations from the previous layer to zero during the forward pass part of the training of the network, see Section 3.3.2. By doing this, the network should be able to better classify the input, even if some activations are missing.

Figure 3.9 shows an example of a training iteration of a neural network when dropout is applied.



**(a)** Standard Neural Net **(b)** After applying dropout.

**Figure 3.9:** Visualisation of dropouts effect on a neural network[1].

#### 3.3.1.5 Fully Connected Layer

The fully connected layer is used to get probabilities for the different classes by looking at what high level features correlates to a particular class the most.

The layer takes the output volume from the layer preceding it as input and outputs a dimensional vector with a size equal to the number of classes the network is designed to classify. For example, if the network is designed to classify digits, the vector would have a size of 10, one element for each digit. Each element in the output vector is connected, with a weight, to each neuron from the previous layer. The layer tries to determine which activations in the previous layer correlates the most to what particular class. This allows for nonlinear combination of features to be used in the classifiers.

#### 3.3.1.6 Softmax Layer

The softmax layer takes the output of the previous network layers and outputs normalized class probabilities. It does this by using a softmax function

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, \ldots, K \tag{3.8}$$

where $K$ is the number of possible outcomes. The softmax layer highlights the largest values while suppressing values that are below the maximum value. The values are logarithmically more suppressed the further below the maximum value they are. The softmax layer is often used in the final layer of a neural network to provide a simple output to the classifier.

#### 3.3.1.7 Activation Function

The activation function part of a neural network is a nonlinear layer that is added throughout the network. This adds nonlinearities to the output of a layer, which is important as most real world data is not linear. Adding nonlinearities also helps to reduce network training times, as they reduce the computational complexity, without a significant decrease in resulting accuracy of the network[22]. They can also help counter the vanishing gradient problem[23] where the lower layers in the network training is very slow due to an exponentially decreased gradient as it moves through the network.

There exist a number of common activation functions that results in different ranges of output. The three most commonly used ones today are the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.9}$$

which limits the input to a range from 0 to 1, the tanh function

$$tanh\left(x\right) = 2\sigma\left(2x\right) - 1 \tag{3.10}$$

where $\sigma$ is the chosen standard deviation. The tanh function fits the input into the $(-1, 1)$ range, or the the so called ReLU function

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} = max\left(0, x\right) \tag{3.11}$$

which only outputs the positive inputs.

The ReLu function is an abbreviation for "Rectified Linear Unit function" and it simply changes negative inputs to zero while forwarding positive real valued inputs, limiting the output to positive values.

### 3.3.2 Training a Neural Network

To have a useful network, the network weights need to be trained so it learns what to look for. This is done through a training process called backpropagation. The training is done by first initializing network weights as small random numbers to promote variation through the network. Then an input is passed through the whole network, a so called forward pass. The first time this is done the output will basically not give any preference to any certain result. This is then passed to a loss function comparing the output to the target. A common loss function is the mean squared error (MSE)

$$E_{total} = \frac{1}{n}\sum_{i}^{n}\left(target_i - output_i\right)^2 \tag{3.12}$$

which, as the name suggest, is the mean of the square of the errors.

Initially this error will be large, and the networks goal is to minimize this error. To minimize the error the different weights contribute to, the error needs to be identified. This is done during the so called backward pass through the network where a stochastic gradient descent is performed by computing the gradients in reverse order (backwards) of the network. Once the derivatives are computed the weights can be updated accordingly so they change in the direction indicated by the gradient. The weight updates has a parameter called the learning rate, $\eta$, that is basically a scalar on the length of each step. A large learning rate may make the model converge in less time, but if the value is too high it could result in steps that are too large for the model to reach an optimal point. Too small of a value and the

model might get stuck in a local optimum that is not close to the global optimum, this is visualized in Figure 3.10.

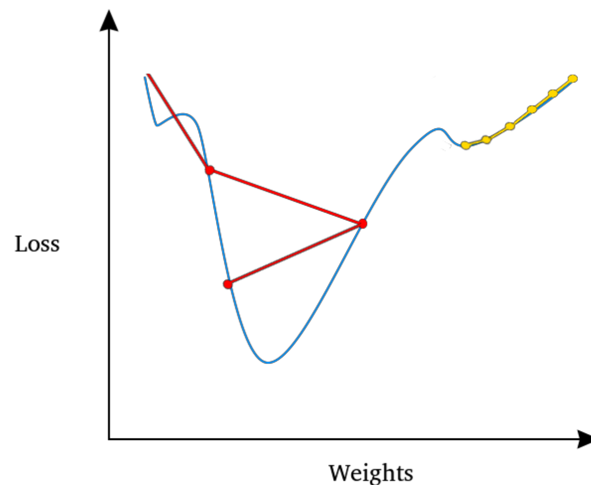



**Figure 3.10:** Visualization of different learning rates. Loss function in blue. High $\eta$ in red that does not reach the global minimum. Low $\eta$ in yellow that gets stuck in a local minimum, and does not reach the global minimum.

There are different techniques to reduce the risk of this, such as using a changing learning rate that starts out large, but gets progressively smaller, or applying a momentum to the gradient to create a tendency to move in the same direction.

#### 3.3.2.1 Training Set

The training set is what the neural network trains and learns from. The neural network adjusts its weights by propagating the error through the network and calculating the accuracy over the training set.

#### 3.3.2.2 Validation Set

By validating the accuracy of the neural network between training iterations against a set that is not used to adjust the weights, it is possible to verify that an increase in accuracy on the training data yields an actual improvement on data not yet seen by the network. If the accuracy of the training data increases while the accuracy of of the validation set decreases the neural network is overfitting to the training set.

#### 3.3.2.3 Testing Set

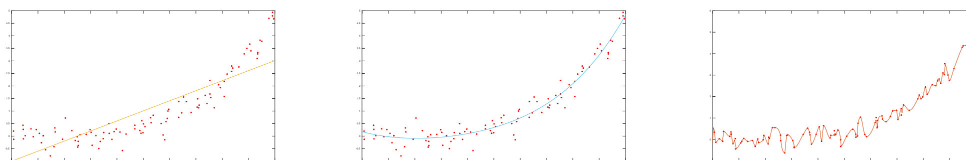
The testing set is used much like the validation set to verify the accuracy of the network. The testing set is however used for verification of the network after training is complete, whereas the validation set is used during training to verify an increase in accuracy.

#### 3.3.2.4 Overfitting

Overfitting[24] is a term that describes a phenomena where a complex model has been made that describes the training data very well, however it does not function well with new data. Such a model tries too hard to explain what can be considered noise in the data, and that results in a model that is generally unfit to use on new data.

In Figure 3.11 there is an example of three different models trying to model the data. Figure 3.11a shows an example of an undertrained model that tries to fit a linear model to nonlinear data. Figure 3.11c shows an example of overfitting, where the model is trying to describe what can be considered as random noise. While the error on the training data is low, when applied on new, previously unseen data, the models predictive performance can be poor.

Neural networks are generally prone to overfitting, and there are many techniques that can help reduce overfitting. Increasing the training set reduces the risk for overfitting, as there is more data to train on. The training data can sometimes be artificially expanded from the existing training set. For example, if the training set is a set of images, new training data can be generated by rotating, scaling, cropping, adding noise or otherwise transforming the existing images. One could also average multiple trained models, provided the resources to produce multiple models exist. Something that is becoming increasingly common is using a dropout layer, such as in the one described in Section 3.3.1.4. In the dropout layer, a portion of the neurons are randomly deactivated, i.e. they are set to 0, during training. This forces the network to have built in redundancies.



**(a)** Underfitting model **(b)** Wellfit model **(c)** Overfitting model

**Figure 3.11:** Visualisation of the concept of under or overfitting a model.

### 3.3.3 GoogLeNet

GoogLeNet, seen in Figure 3.12, was the winning architecture in ImageNets "Large Scale Visual Recognition Challenge" of 2014[25] when they introduced the so called Inception module. It was one of the first architectures strayed from the classical approach of sequentially stacking layers. It adds multiple parallel layers in sections called Inception modules. This can be seen in Figure 3.12.

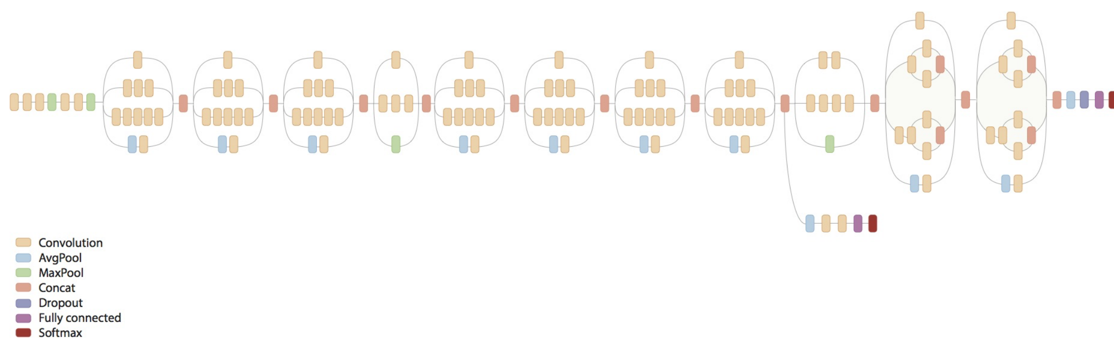
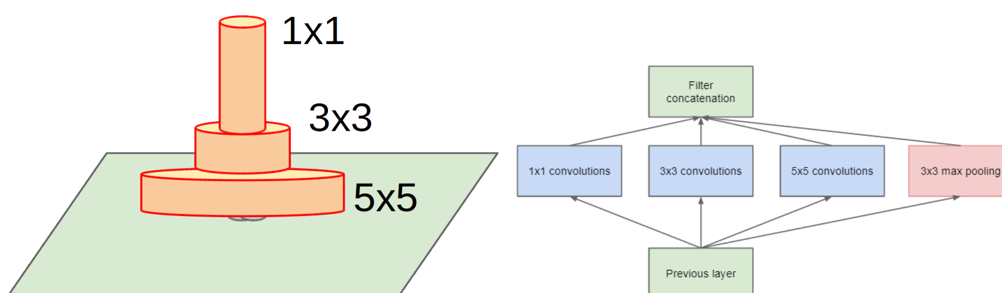



**Figure 3.12:** Overview of the GoogLeNet Inception v.3 architecture.

#### 3.3.3.1 The Inception Module

The GoogLeNet is built with the idea that in images, correlations tend be be both local, and be slightly spread out around these areas. To catch this filters of different sizes, ($1\times1$, $3\times3$ and $5\times5$) are added in parallel to cover larger areas, while keeping a fine resolution for the smaller information in the images, as done in Figure 3.13a. To summarize the content from the previous layer, a pooling layer is added. Figure 3.13b shows this as the naïve idea of the Inception module.




**(a)** The naïve idea to stack filters of different sizes. **(b)** Inception module, naïve version.

**Figure 3.13:** The Inception module[2].

However, doing it this way will not work as it leads to an explosion of computations and outputs. The solution to this is to add a $1 \times 1$ CONV operation before the $3 \times 3$ and the $5 \times 5$ layer, one is also added after the pooling layer, so the full Inception is as showed in Figure 3.14. This provides a dimensionality reduction to keep the computational cost of the network down. ReLu layers are added after each CONV to add nonlinearities to the network.

Stacking these modules allows for simple managing of parameters when designing the network, as each module can be individually tweaked.

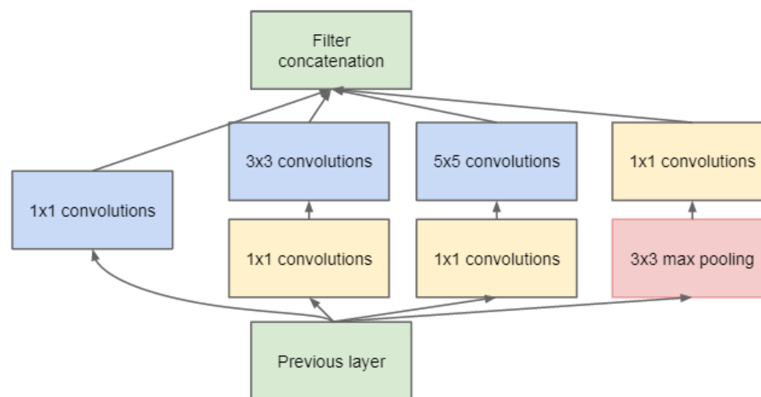



**Figure 3.14:** Inception module with dimension reductions[2].

# 4

# Method

This chapter describes how the work during the thesis was carried out in order to obtain the results. The objectives described in Section 2.2 consist of several subproblems that all has to be solved in order to achieve the main goal. These subproblems were the base of how the work was carried out. The first approach was an iterative process of building algorithms and testing solutions and then testing against GUIs of increasing complexity. Initial tests were performed on a simple, simulated GUI to ensure full control of variables during early stages of development. A conceptual view of the system as shown in Figure 4.1 was used as an initial model.

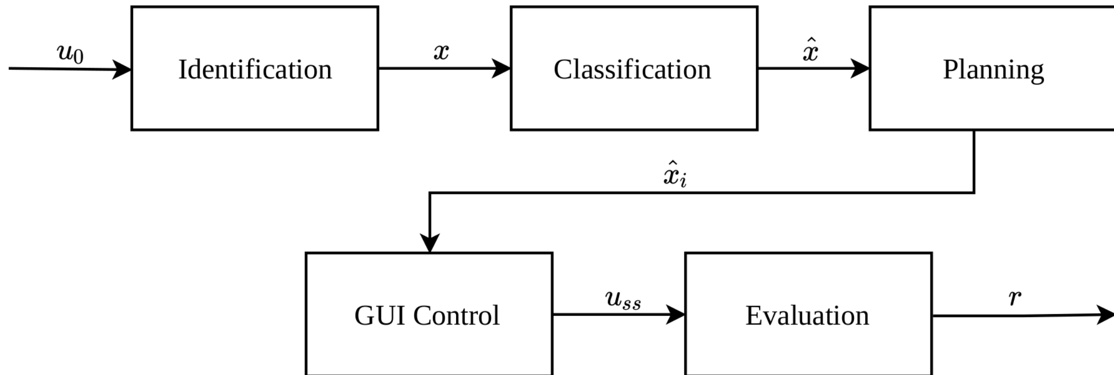



**Figure 4.1:** Schematic view of the early system.

The data that flows through the system is referred to as:

$u_0$ The initial image from the GUI

$x$ Identified areas of interest

$\hat{x}$ Classified objects

$\hat{x}_i$ Selected object to try an interaction on

$u_{ss}$ Steady state image of GUI

$r$ Result of comparison.

## 4.1 Tools

The development process has seen use of various third party tools, which are described briefly here.

### 4.1.1 MATLAB

Initial prototyping of the edge detection algorithm was done using MATLAB since it offers off the shelf implementations of many methods such as Gaussian and Sobel filtering. However, in the case of the filtering, these operations were written for the sake of this thesis to be able to control pixel padding and performance.

### 4.1.2 Android Device

The Android device that the solution was developed on was a Sony Xperia L C2105 running CyanogenMod 12-20150616-NIGHTLY-taoshan, Android 5.0.2[26]. The device has a resolution of $480 \times 854$, and has three virtual navigation buttons (BACK, HOME and APP_SWITCH) replacing common hardware navigation buttons.

### 4.1.3 Android Studio

A test environment for Android has served as a convenient way to test the algorithms on an actual touch screen. At first, the idea was to test the algorithms in a simulated environment with full control of interactive events. This proved unnecessary since it would basically be the same as running the tests on a phone hooked up to Android studio.

#### 4.1.3.1 Android Debug Bridge

Android Debug Bridge (adb) is a command line tool for communication between an Android device and a PC. The device can either be emulated or real. The adb command tool includes three main components:

**A client** that sends commands to the device via a command line terminal.

**A daemon (adb)** which runs the actual commands on the connected device.

**A server** which manages communications between the client and the daemon.

This is the main communication protocol used to control and navigate the Android

device. A complete list of the adb related commands used in the project can be found in Appendix B.2.

### 4.1.4 Python

Python was used throughout the thesis duration and the final solution was written in Python 3.5.2[27]. There are a set of required packages that were used in the program, these are listed and briefly described in Appendix B.1.

### 4.1.5 TensorFlow

TensorFlow is an open source library for machine learning. It is used extensively by Google to perform speech recognition, recommendation features and image classification. There is a Python API for TensorFlow that can help during development. TensorFlow is the base on which TensorBox rests.

#### 4.1.5.1 TensorBoard

TensorBoard is a tool accompanying TensorFlow to visualize the process of training neural networks. This is very helpful when training for a long time (several days). If the accuracy does not seem to approach expected levels after a couple of hours, it gives an indication that something might be wrong and it is possible to abort the training process and start over with better training data or tuned parameters.

#### 4.1.5.2 TensorBox

TensorBox is a framework based on TensorFlow for training neural networks to detect multiple objects in images. The basic configuration implements the simple and robust GoogLeNet-OverFeat algorithm. Furthermore, the framework also provide an implementation of the ReInspect algorithm, reproducing state of the art detection results in highly occluded scenes. However, this is not in the scope of this thesis since the GUI is assumed to be completely visible in its entirety.

The final version of the resulting neural network uses a modified version of TensorBox. Out of the box, TensorBox comes with a rather deficient tool for annotating the training data in an orderly fashion. A new tool to fill that need has since been created.

## 4.2 Data Acquisition

In order to solve any problem, data is needed, more data is better, even more so when using machine learning.

Some data has been collected by the means of taking images of phones GUI with a camera, and taking screenshots with adb. But as the aim was for a general solution and the limited access to different GUI, the main source of image data has been the Internet with its wide array of open source images.

## 4.3 Identification

Since one of the aims of the thesis was to explore the possibility to use image analysis techniques to distinguish between interesting areas of a GUI, two possible methods were explored. The first approach was using sobel edge detection to try and find objects. Buttons and other interactive elements tend to have sharp borders that run continuously around the interactive area which indicated that edge detection would be a suitable solution. The other method was identification using a neural network. It was deemed an interesting alternative after reading about TensorFlow and TensorBox in particular.

### 4.3.1 Edge Detection Method

Starting with an image of the GUI, the edge detection was done in several steps, namely grayscale conversion, seen in Figure 4.2, Gaussian filtering, seen in Figure 4.3, Sobel filtering, seen in Figure 4.4 and finally boundary detection, seen in Figure 4.5. Since the goal is to find intensity gradients, the color information was deemed unnecessary.

**Figure 4.2:** Before and after grayscale converison.

Gaussian filtering is necessary to get rid of false positive edges. That is, edges that occur due to noise or other high frequency components in the image. The size of the Gaussian filter was calibrated manually using trial and error. A method to choose the filter size dynamically was considered but never finished due to the decision to abandon the method entirely in favour of a neural network.

**Figure 4.3:** Before and after Gaussian filtering.

When the image has been smoothed using a Gaussian filter, the intensity gradients are calculated using a Sobel process. As seen in Figure 4.4, the process yields two images that are combined.

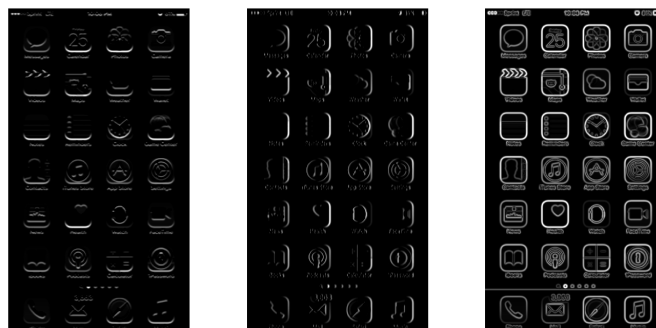


**Figure 4.4:** The results of Sobel filtering ($G_x$, $G_y$ and $G$).

Using the result of the Sobel filtering, bounding areas are found by identifying edges that enclose an area above a certain threshold. Thus, the object identification process using edge detection is finished.

**Figure 4.5:** The results of Boundary detection.

### 4.3.2 Identification Using TensorBox

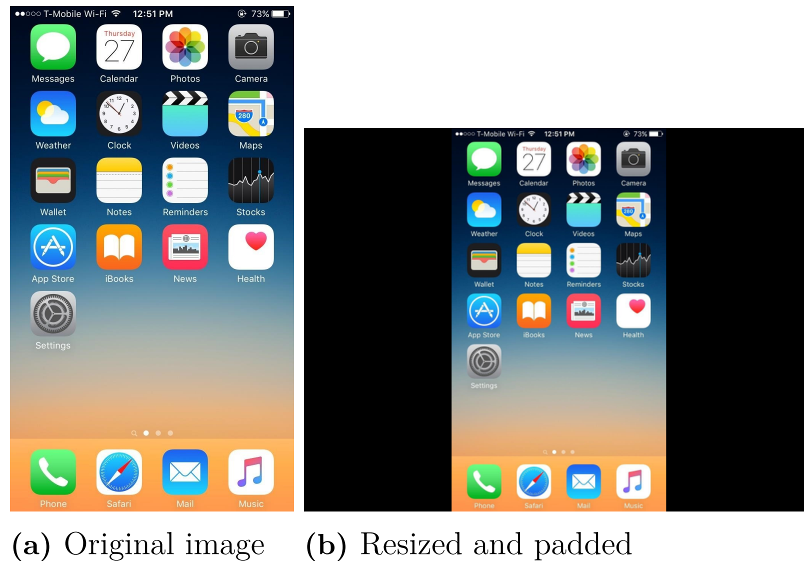
By using a neural network to identify objects in a GUI the majority of the work is spent finding suitable images to use as training data. This meant gathering as many images of Android and iOS GUIs as possible. The rather time consuming task of manually annotating the images (selecting the areas which the neural net should find once trained) led to a rather small training set of 62 images. Before any meaningful results came out of this method, a lot of time was spent preparing the development environment to accommodate for running TensorBox on the GPU.

#### 4.3.2.1 Preparing Training Data

Before training can begin, it is necessary to collect and prepare the data to be used when training. Collecting images of GUIs is simple enough. By wielding the mighty sword that is Google Image Search, a collection of images of GUIs quickly began to take form.

On the matter of preparing the data, TensorBox does actually include some means of preparing training data. However, the application is slow and clunky. That is why a new way of preparing the data had to be devised. Firstly, in order for TensorBox to efficiently process the images, they need to be scaled to an appropriate size. If the resolution of the images differs from the correct resolution (640x480), they are scaled and/or padded with black pixels, as shown in Figure 4.6. Next, the manual labour of selecting the areas that are considered clickable has to be done with relative speed. Thus, an application to interactively annotate the images was created. The result of using this application is a JSON-object, which is the format used throughout TensorBox to catalogue training data.

The tool to aid the annotation process (manually selecting which areas the trained neural network should find) is simple but effective. First, the directory of the training




**(a)** Original image **(b)** Resized and padded

**Figure 4.6:** Training image before and after resizing.

images is selected. The tool then searches for a specific JSON file. If the file exist, the tool finds the images that has not yet been annotated and the process is continued. If the JSON file is not present, it is created and the process is started. When annotating an image, the user marks two opposing corners by clicking twice (Figure 4.7a-Figure 4.7b). This yields four coordinate pairs, $(x_1, y_1)$, $(x_2, y_1)$, $(x_1, y_2)$ and $(x_2, y_y)$, one pair for each corner of the rectangle. The process is then repeated as many times as needed for each image, as shown in Figure 4.7c.

**(a)** Step one: choose first corner. **(b)** Step two: choose second corner. **(c)** Annotating finished.

**Figure 4.7:** The three steps of the annotation process.

When all images are annotated, the JSON-file is passed through a splitting function that randomly selects a predefined ratio (50% in our case) of images to be included in the training set and the validation set.

With two sets of prepared data, the training can begin. The process was performed as described in Section 3.3.2. Several runs were performed using the Overfeat-Rezoom method which produced a file containing the weights of the network. These weights were then used to evaluate new images using this method.

## 4.4 Classification

Areas in the GUI was divided into one of two categories, namely an active area or a passive area. If the user can interact with a given area, it will be classified as active, or interactive. Otherwise it will be labelled as passive. Verification that the identified objects were interactive was done by comparing the screens before, and after, an action on a specified coordinate. If the screens differ above a threshold, the coordinate will be classified as active.

The comparison was made using perceptual hashing. Only the dHash hashing algorithm as described in Section 3.1 was used. The dHash algorithm was chosen due to its simplicity to implement in Python, its relative speed and its relative accuracy.

The scaling size of the image was chosen to be $9 \times 8$, with a Hamming distance threshold of one or below to indicate a matching image. This to accommodate for small changes, such as the clock changing digits, which should not affect identity of the state.

## 4.5 System Modelling

To enable path planning, memory and other features in the system, a model of the system is critical. Since most GUIs could be considered discrete systems, a solution where the system is modelled as a state machine with a graph structure is created.

By using a state model, states that are functionally identical can be marked as the same and thus redundant states are removed. This could be useful for example if two icons have switched place, two icons link to the same function, or that the aesthetics of the GUI has been altered, but the individual functions remain the same.

The individual screens of the GUI are modelled as states and nodes are created for each state. Interactive elements are added as transitions between these states.

The state graphs were generated with the DOT language via the python library Transitions[28].

## 4.6 GUI Control

To navigate and control the GUI, a means of interacting with the hardware was needed. This was done via a PC by sending commands from a python script via adb to the device. With adb a lot of different touch event types relevant for this work can be simulated.

## 4.7 Proposed Method

The proposed method is a combination of using the GoogLeNet-OverFeat algorithm in TensorBox to come up with a set of coordinates for the estimated interactive objects and using perceptual hashing to verify that the object is interactive. That is if clicking on the predicted coordinate produces a new screen.

### 4.7.1 The Algorithm

The proposed method starts out by grabbing an image of the initial screen and calculating the hash value of the image. It then runs the image through the trained network to acquire a set of coordinates that the network believes interactable objects are located at.

All these acquired coordinates are then acted upon in order, performing a breadth first search. After each action on a coordinate, a new image is taken, and a new hash is calculated. If this new hash is different from the previous hash, the coordinate that was acted on is assumed to be interactive.

If this image hash pair is new and not part of a previous set it is assumed to be a new state, and a new node is created with a transition to the previous node. If the image hash pair is part of a previous set, a transition is created between that node and the previous node.

A flowchart over the proposed system can be seen in Figure 4.8. A more detailed state machine over the system can also be found in the Appendix, Figure A.2.

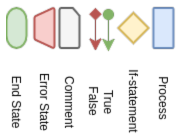



**Figure 4.8:** The proposed workflow.

# 5 Results

## 5.1 Identification

In this section, the results of identifying GUI objects using edge detection and a neural network respectively are presented.

Initially, the edge detection approach showed great promise. In the first test case which consisted of an iOS home screen with a solid black background, the edge detection method performed consistently well when tuned. Some false positives, as seen in the top of Figure 5.1a, appeared, as well as some missed icons (*Videos*, *Calculator* and *Safari*). This was due to a tuning problem that resulted in more false positives if the sensitivity was increased to catch all icons.

| | Hits | False Positives |
|---|---|---|
| Edge Detection | 24/28 (85.7%) | 3 |
| Neural Network | 27/28 (96.4%) | 6 |

**Table 5.1:** Identification results, plain background.

From Table 5.1 and Figure 5.1, we can conclude that in the simple case of identifying objects against a plain background, the edge detection method was not as accurate as using the neural network, but it resulted in fewer false positives. These false positives would not cause a lot of headache when running an automated test of a touch screen since those areas are deadzones.

Moving on to a slightly more difficult test case. Here, the background has a slight gradient, as seen in Figure 5.2. There are no sharp edges in the background, but the edge detection method is rendered completely useless when using the same tuning as the previous example. The algorithm is simply too sensitive for color fluctuations in the background. This is partly due to the loss in color depth when converting the image to black and white. The algorithm is designed to use too few levels of intensity. On the bright side, the neural network performs almost perfectly with only one missed object here as well.

As a final test of the neural network, a GUI with icons of varying size and shape

(a) Using the edge detection method (b) Using the neural network

**Figure 5.1:** Identified icons on an iPhone GUI with a plain background.

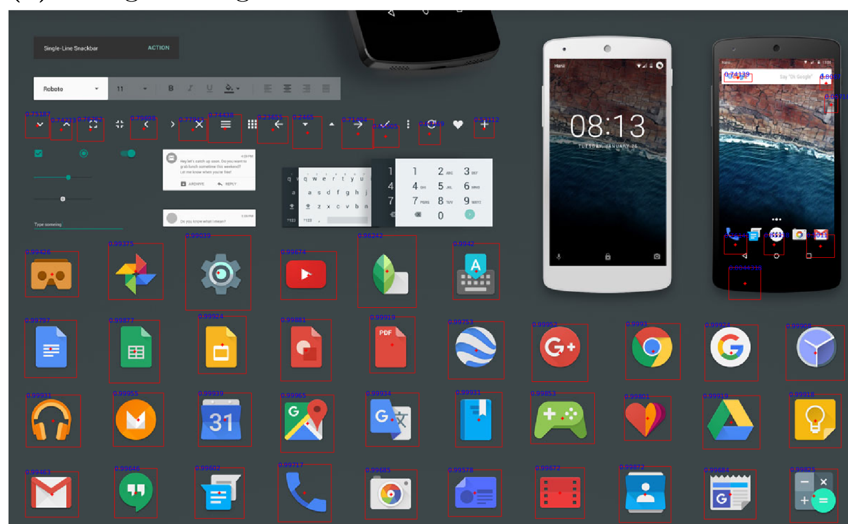
(a) Using the edge detection method (b) Using the neural network

**Figure 5.2:** Identified icons on an iPhone GUI with a background that has a color gradient.

was tested. Again, the edge detection method is useless while the neural network

manages to find all application icons of normal size, as well as some of the smaller navigational icons (arrows and such). Note that none of these particular images were included in the training set. The result of these can be seen in Figure 5.3.

**(a)** Using the edge detection method




**(b)** Using the neural network

**Figure 5.3:** Identified icons on a generic image with a slight gradient.

The net was trained on a small dataset of only 62 images, with a resulting accuracy of over 95% on the validation set was which was achieved after only about 4300 iterations of training, which was about 10 minutes on the computer used for training.

The computer used was fitted with 16 GB of RAM, an Intel Core i7-6700HQ processor with a clock frequency of 2.6 GHz. During training, the Nvidia GeForce GTX 960M graphics adapter handled much of the computations.

The reason for the small dataset was a combination of not prioritizing the time-consuming work of annotating the images combined with the realization that a

decent result were achieved anyway. That the network was pretrained also helped reducing the training time required.

## 5.2 Classification

Table 5.2 shows the Hamming distance values of the dHashes generated from the images in Figure A.4 with the images scaled to $7 \times 6$. The algorithm turns the images into grayscale before the hash is calculated, the grayscale images can be seen in Figure A.5. The Hamming distance is commutative, thus the distances below the diagonal are not shown since they are mirrored across the diagonal. The cells marked in blue are correctly identified as the same image ($d_h = 0$), cells marked in yellow are images with a low hamming distance ($d_h \leq 3$) that might run the risk of being classified incorrectly. The cells marked in red are image pairs that were classified as the same image incorrectly ($d_h = 0$).

**Table 5.2:** Hamming distance of the dHash values between the images in Figure A.4, downscaled to 7x6

| | **1** | **2** | **3** | **4** | **Grey** | **Red** | **Blue** | **Green** |
|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 2 | 0 | 3 | 7 | 8 | 8 | 7 |
| **2** | | 0 | 2 | 3 | 7 | 8 | 8 | 7 |
| **3** | | | 0 | 3 | 7 | 8 | 8 | 7 |
| **4** | | | | 0 | 8 | 8 | 8 | 8 |
| **Gray** | | | | | 0 | 1 | 1 | 0 |
| **Red** | | | | | | 0 | 0 | 1 |
| **Blue** | | | | | | | 0 | 1 |
| **Green** | | | | | | | | 0 |

The results of a larger image scaling, $11 \times 10$, can be seen in Table 5.3 with the same colour code as in the previous table.

**Table 5.3:** Hamming distance between the dHash-values of the images in Figure A.4, downscaled to 11x10

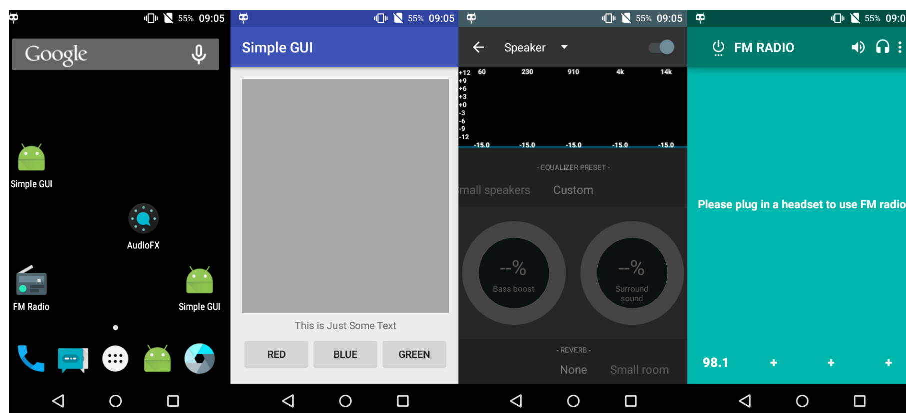
| | **1** | **2** | **3** | **4** | **Gray** | **Red** | **Blue** | **Green** |
|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 4 | 3 | 9 | 19 | 20 | 19 | 20 |
| **2** | | 0 | 7 | 9 | 20 | 20 | 20 | 20 |
| **3** | | | 0 | 11 | 19 | 20 | 19 | 20 |
| **4** | | | | 0 | 20 | 20 | 19 | 19 |
| **Grey** | | | | | 0 | 3 | 4 | 5 |
| **Red** | | | | | | 0 | 3 | 3 |
| **Blue** | | | | | | | 0 | 3 |
| **Green** | | | | | | | | 0 |

## 5.3 Test Runs

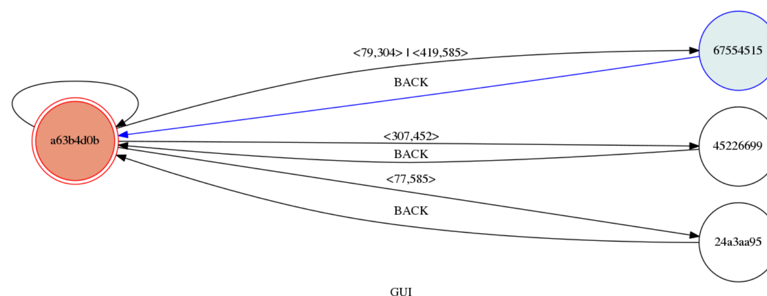
In this section two test runs of the complete algorithm will be presented.

The first is a very simple run, with only four states and four tap transitions. The result of this can be seen in Figure 5.4. The program was allowed to run to completion from the start screen with access to four application icons, where three are unique. None of the applications contained sub-functions.

The second run contains more advanced applications, with a multitude of states and transitions. These can be seen in Figure 5.5. The program was run from a start screen with access to three application icons, where all three were unique. One of the applications was a gallery with complicated substructures with several links to the same image. The process was terminated early as the gallery contained over two hundred images at the point of the test.

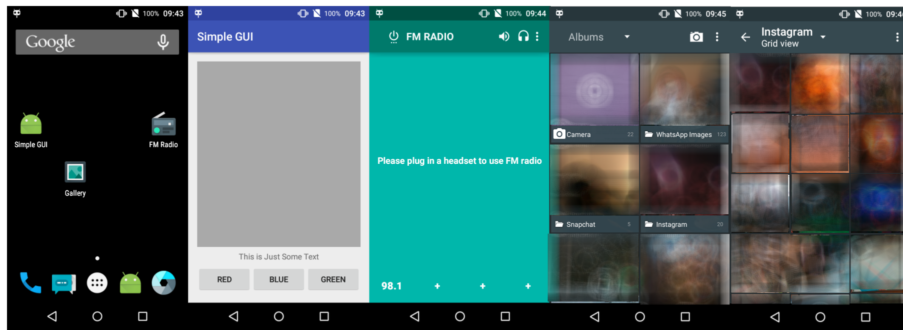



**(a)** The four identified states

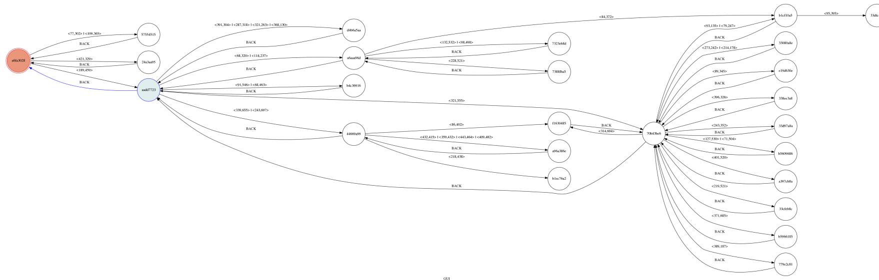



**(b)** Resulting State Machine.

**Figure 5.4:** Completed run with simple applications. The node names in the figure are the hashing values of each state.




**(a)** Five excerpts from the identified states. The images in the gallery has been blurred for the report. The node names in the figure are the hashing values of each state.



**(b)** Resulting State Machine. Larger version can be found in the Appendix, Figure A.3.

**Figure 5.5:** Early terminated run with one advanced application.

# 6

# Discussion and Concluding Remarks

Upon building the system using machine learning techniques, discussions among eachother, as well as with the supervisor at Benchnode Technology AB, Mikael, led to some concerns regarding just how useful such a system might be in a real world testing situation.

As efficient as it may be to identify objects using a neural network, the system cannot guarantee 100% coverage. This is a major flaw since it is quintessential to be able to depend on your testing methods when testing a novel system. However, it is not hard to imagine that such a system would save a lot of time in an otherwise manual process.

The feature that allows a GUI to be visualized as a graph shows more promise. When ordering a new interface to a touch screen, the automotive industry has to make sure that the driver is not distracted by an overly complicated interface. Requirements such as *"the graphical user interface should be intuitive and easy to navigate"* is not very descriptive and is prone to be misinterpreted. An alternative, less ambiguous way of conveying the desired result could be to reason in terms of interface depth. This kind of measurement is quickly externalized if the GUI is represented as a graph. Now, the requirement could instead be written in terms of maximum depth for a given function.

As for the choice to use TensorFlow, in retrospect, the time invested in understanding the tool was worth it. However, it could easily have been a not very pleasant experience if the results were not as good. Had the solution been developed on a Windows machine, as it started out, the experience might have been worse. TensorFlow is much easier to use on a Linux based system. As such a transition from Windows to Linux was made in the middle of the thesis work. This transition led to a slower development process but in the end, the choice to abandon Windows was rewarding, both in terms of results, but also for us as developers. It also turned out to be a good choice to abandon the edge detection method at a relatively early stage when it failed to produce good results on relatively simple GUIs instead of spending time developing it further. In the end, it was deemed to be a far too daunting task to make the edge detection method general enough. This is due to the problems

to tune the Gaussian filter in order to make the method work on varying kinds of backgrounds.

Our neural network cannot be considered very general, as the number of training images are very low. However, as a proof of concept, it does the job, but to truly be certain that this is indeed an accurate method, it should have been tested on a variety of GUIs, as well as different kinds of touch screens.

Perceptual hashing was proved to be a very useful tool in the image analysis toolbox. dHash performed unexpectedly good, and was very easy to change its specificity to how small or large the differences between the images you wanted to catch. It also acted as a verification of the identified elements that the neural network identified, as it found false positives. However it will not be helpful in finding the elements the neural network missed. Missed elements is harder to compensate for than false positives and it is something that needs to be minimised.

## 6.1 Further Development

The machine learning approach to identify objects in a GUI could be expanded to handle classification of different types of areas, not only icons, but for example menues, sliders or other GUI functions. This was something that was discussed in the planning phase, but left out due to shortage of time. It would also be interesting to investigate the possibility to include character recognition and natural language detection.

Another feature that was not studied is the possibility to evaluate how the GUI is responding to input. That is, a system to compare how the GUI is responding compared to a known response. This would be interesting when performing regression tests on a GUI that is under development.

Naturally, there is always room for improvement when it comes to the neural network. As mentioned in the previous section, our network is not to be considered very general. It performs well in our testing environment but to be able to work in real world testing scenarios, the training set would have to be expanded significantly.

Different hashing algorithms could be applied in unison with each other depending on what is needed. For example the dHash algorithm could be used initially to find very similar images, and then another more accurate, but computational heavier, algorithm could be used, such as the DCT based hashing algorithm pHash. In our case this was not needed, but in a large scale set up where the input images aren't as pristine, and the volume is larger, computational time might be of interest.

As of now, the only measurement of the GUI is depth. For the accuracy and efficiency of the testing system to be measured completely, something more to compare the results against should be considered. The accuracy could be verified by comparing

the result of interacting with an object to a screenshot that shows the correct result. Efficiency could be the time it takes to execute the testing procedure.

Another kind of quality measurement could instead be performed on the GUI itself. The score could be calculated on parameters like the euclidean distance travelled on the screen to arrive at a certain menu, time spent navigating the menus, the number of button presses and so on.

To take the GUI evaluation further, the concept of usability could be introduced. It is a highly subjective concept and therefore difficult to measure. There are however some design standards for interactive media that could be checked. For example, the color green often indicates an affirmative behaviour, and red is the stopping color. It is also often confusing for the user if the *OK*-button is to the right of *CANCEL* when presented with a standard dialog box. Other things that could affect the general usability is button size, the presence of descriptive text or the number of objects appearing on the screen simultaneously. These examples are all making the screen appear as cluttered and distracting, which is not desired when navigating a GUI at the same time as doing something as inherently dangerous as driving at the same time.

The solution could be made truly general and usable in a real world testing scenario if it is possible to apply to any touch screen GUI. As of now, it is only possible to test Android tablets and phones. Consider a physical rig that is mounted above a screen. This rig could use a camera to grab images of the GUI, and a pointing device acting as a mechanized finger to interact with the screen. This would yield a truly general testing solution. It would also introduce a new set of interesting problems to solve, both in the field of mechanical construction as well as handling image quality issues and distortions such as screen glaring.

# A

# Appendix 1

## A.1 Figures

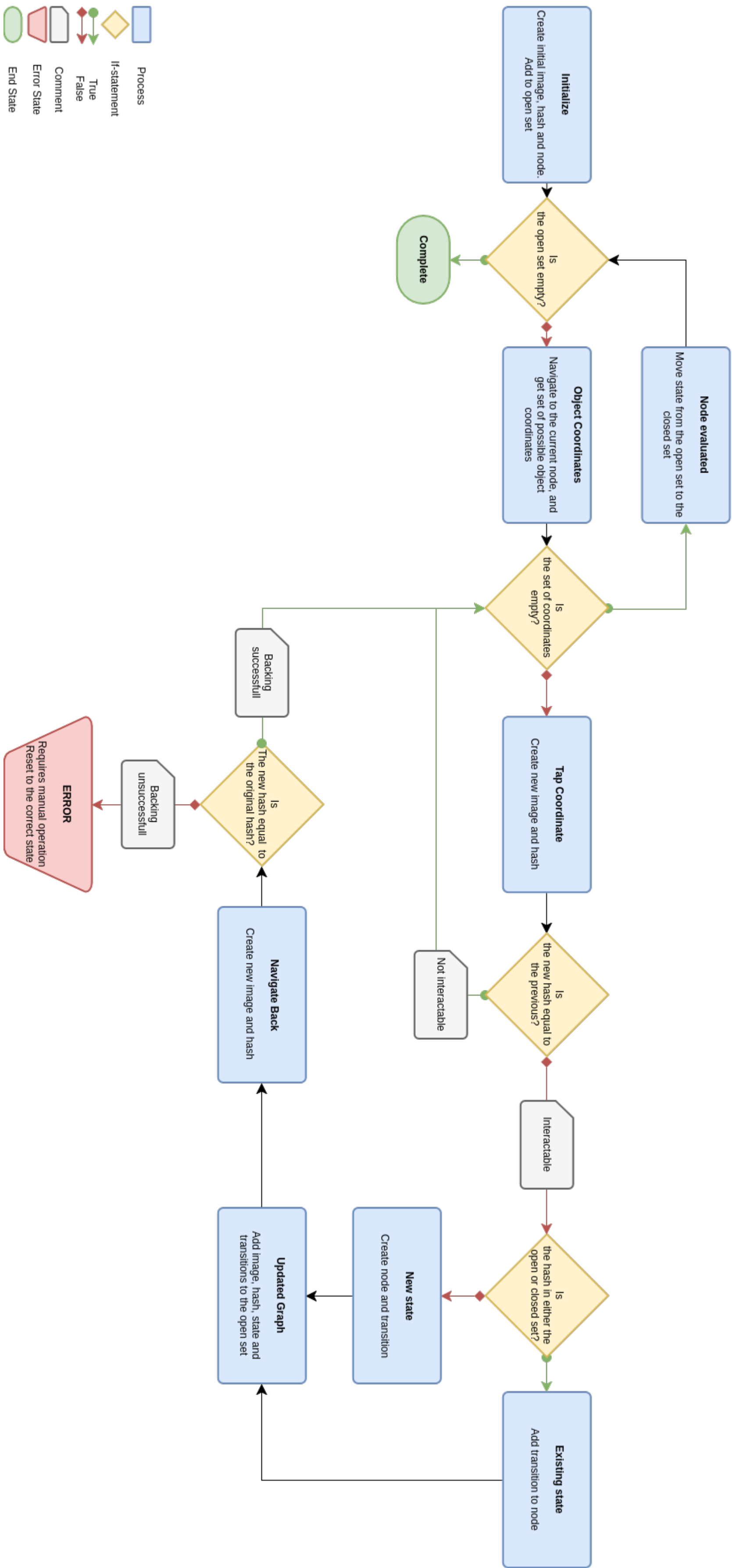



**Figure A.1:** The proposed algorithm

**Figure A.2:** State diagram and transitions over the used program

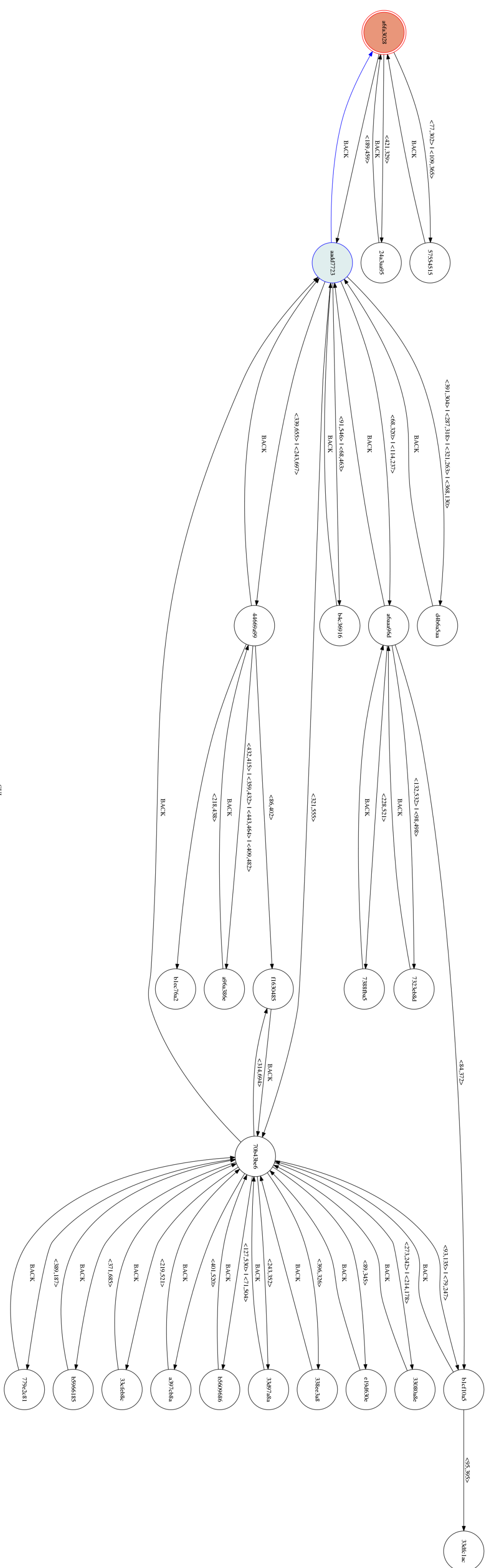
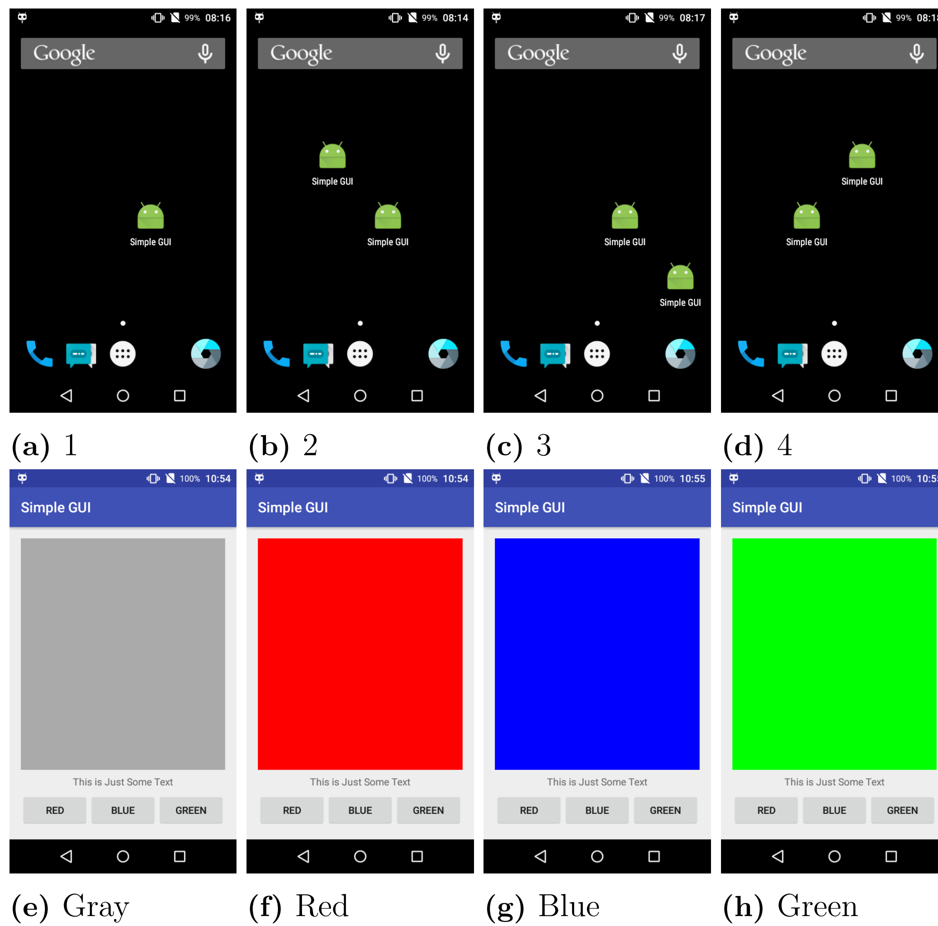


**Figure A.3:** Resulting state machine of a early terminated run on a complex GUI structure.

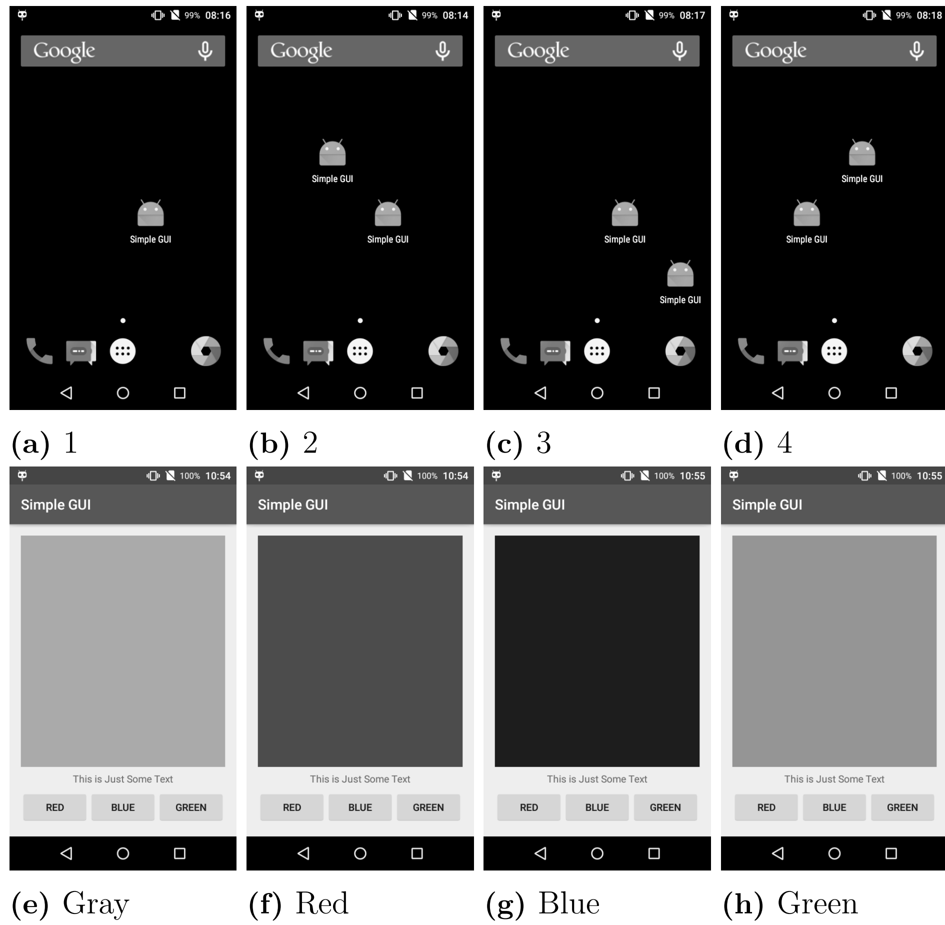
## A.2 Hashing comparisons




(a) 1 (b) 2 (c) 3 (d) 4

(e) Gray (f) Red (g) Blue (h) Green

**Figure A.4:** Images used to test perceptual hashing algorithms.




(a) 1 (b) 2 (c) 3 (d) 4

(e) Gray (f) Red (g) Blue (h) Green

**Figure A.5:** Images in grayscale that were used to test perceptual hashing algorithms.

# B

# Appendix 2

## B.1 Python Packages

- fileinput[29]: This module implements a helper class and functions to quickly write a loop over standard input or a list of files.
- glob[30]: Unix style pathname pattern expansion
- itertools[31]: This module implements a number of iterator building blocks providing tools for fast and efficient "iterator algebra".
- json[32]: JSON encoder and decoder.
- math[33]: Provides access to the mathematical functions defined by the C standard.
- os[34]: Miscellaneous operating system interfaces
- PIL[35]: Imaging Library
- re[36]: This module provides regular expression matching operations
- subprocess[37]: Allows spawning of new processes, connections to their input/output/error pipes, and their return codes.
- sys[38]: System-specific parameters and function
- time[39]: Time access and conversions
- tkinter[40]: Provides a robust and platform independent windowing toolkit
- transitions[28]: A lightweight, object-oriented state machine implementation
- warnings[41]: Warning control

## B.2 Android Debug Bridge: Code samples

**Listing B.1:** List all connected devices

```
1 adb devices
```

**Listing B.2:** Take a screencapture and move it to "PATH"

```
1 adb -s deviceID exec-out screencap -p > "PATH"
```

**Listing B.3:** Send the keyevent for the HOME button

```
1 adb -s deviceID shell input keyevent KEYCODE_HOME
```

**Listing B.4:** Send the keyevent for the BACK button

```
1 adb -s deviceID shell input keyevent KEYCODE_BACK
```

**Listing B.5:** Send the keyevent for the SWITCH button

```
1 adb -s deviceID shell input keyevent KEYCODE_APP_SWITCH
```

**Listing B.6:** Perform a "tap" at location X Y

```
1 adb -s deviceID shell input tap X Y
```

**Listing B.7:** Perform a "swipe" from X Y to X Y during T

```
1 adb -s deviceID shell input touchscreen swipe X Y X Y T
```

# Bibliography


[1] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 227–236, 2014.

[2] L. A. dos Santos, "Googlenet." leonardoaraujosantos.gitbooks.io/artificial-inteligence/, 2014. [Online; accessed 05-May-2017].

[3] V. (OpenSource), "Linux desktop testing project," 2016. [Online].

[4] X. Fang, B. Sheng, P. Li, D. Wu, and E. Wu, "Automatic gui test by using sift matching," *China Communications*, vol. 13, no. 9, pp. 227–236, 2016.

[5] Exforsys, "What is monkey testing," 2017. [Online; accessed 2017-01-24].

[6] T. Daboczi, I. Kollar, and G. Simon, "How to test graphical user interfaces," *IEEE Instrumentation & Measurement Magazine*, 2003.

[7] Wikipedia, "Scale-invariant feature transform — wikipedia, the free encyclopedia," 2017. [Online; accessed 25-April-2017].

[8] Wikipedia, "Image histogram — wikipedia, the free encyclopedia," 2017. [Online; accessed 25-April-2017].

[9] M. Özuysal, M. Calonder, V. Lepetit, and P. Fua, "Fast keypoint recognition using random ferns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2009.

[10] Wikipedia, "Perceptual hashing — wikipedia, the free encyclopedia," 2017. [Online; accessed 25-April-2017].

[11] N. Krawetz, "Kind of like that." The Hacker Factor Blog, 2013. [Online; accessed 04-April-2017].

[12] B. Yang, F. Gu, and X. Niu, "Block mean value based image perceptual hashing," *2006 International Conference on Intelligent Information Hiding and Multimedia*, pp. 1–4, 2006.

[13] N. Krawetz, "Looks like it." The Hacker Factor Blog, 2011. [Online; accessed 04-April-2017].

[14] E. Duda, "Image hash comparison — github," 2015. [Online; accessed 20-April-2017].

[15] E. Klinger and D. Starkweather, "phash - the open source perceptual hash library," 2010. [Online; accessed 2017-02-22.

[16] Wikipedia, "Discrete cosine transform — wikipedia, the free encyclopedia," 2017. [Online; accessed 19-April-2017].

[17] C. Zauner, "Implementation and benchmarking of perceptual image hash functions," Master's thesis, FH Upper Austria, 2010. [Published on phash.org].

[18] Wikipedia, "Hamming distance — wikipedia, the free encyclopedia," 2017. [Online; accessed 25-April-2017].

[19] Z. Blanco, "Neural networks and the backpropagation algorithm," 2015. [Online; accessed 2017-05-18.

[20] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *European Conference on Computer Vision*, vol. 1, no. LNCS 8689, pp. 818–833, 2014.

[21] A. Karpathy and J. Johnson, "Understanding cnn," 2017. [Online; accessed 2017-01-24.

[22] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (J. Fürnkranz and T. Joachims, eds.), pp. 807–814, Omnipress, 2010.

[23] Wikipedia, "Vanishing gradient problem — wikipedia, the free encyclopedia," 2017. [Online; accessed 22-May-2017].

[24] Wikipedia, "Overfitting — wikipedia, the free encyclopedia," 2017. [Online; accessed 17-May-2017].

[25] ImageNet, "Large scale visual recognition challenge," 2014.

[26] Taoshan, "Cyanogenmod 12," 2015. [Online; Discontinued].

[27] Python, "Python 3.5.2," 2016. [Program Language].

[28] T. Yarkoni, "transitions," 2017. [Library package].

[29] Python, "fileinput," 2017. [Library package].

[30] Python, “glob,” 2017. [Library package].

[31] Python, “itertools,” 2017. [Library package].

[32] Python, “json,” 2017. [Library package].

[33] Python, “math,” 2017. [Library package].

[34] Python, “os,” 2017. [Library package].

[35] Python, “Pil,” 2017. [Library package].

[36] Python, “re,” 2017. [Library package].

[37] Python, “subprocess,” 2017. [Library package].

[38] Python, “sys,” 2017. [Library package].

[39] Python, “time,” 2017. [Library package].

[40] Python, “tkinter,” 2017. [Library package].

[41] Python, “warnings,” 2017. [Library package].