Adversarial attacks on modern deep facial recognition systems using Deepfool

Ioannis Keravnos

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Artificial Intelligence
of the
University of Aberdeen.

Department of Computing Science

2021

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2021

Abstract

Adversarial attacks on machine learning models trained for image recognition tasks have long been studied. Common practices include various perturbations on the input to induce a wrong prediction, while being almost undetectable to the human eye. One such strategy is deployed by Deepfool, an adversarial attack based on linear approximation of decision boundaries calculated from gradients. The algorithm's main objective is to repeatedly use this linear approximations of decision boundaries to calculate the minimal perturbation which projects an image to the closest wrong class boundary, thereby 'fooling' the system.

In the context of image recognition and for this study specifically facial recognition, the rise of new robust state-of-the-art architectures makes adversarial attacks even more difficult. Adding to this, while Deepfool offers a good approach in that regard, other attacks have also demonstrated better performance recently.

This allows for the exploration of a few research questions regarding Deepfool attacks and facial recognition models. This paper aims to analyse the possibility of extending Deepfool to create different variants and to measure their performance on state-of-the-art architectures by calculating their adversarial robustness. As a result we propose two attack variants in order to increase the performance of the attack. An evaluation of the two variants against the original is carried out on three different model architectures on a subset of the VGGFace2 dataset.

Keywords: Adversarial examples, DeepFool, Robust machine learning, Face recognition

Acknowledgements

Hardly any great feats are ever achieved alone. As such, it would be extremely unfair if credit was not given to everyone that helped in the process of this research and everything achieved this academic year. Without their unwavering support and efforts, none of this would be possible.

I would like to first acknowledge the massive support and guidance of my project supervisor Matthew Collinson, who provided me with so much and always did so with a cheerful attitude. His enthusiasm as well as academic expertise were fundamental in accompanying me until the end of this project in this difficult year. I am filled with gratitude for all his advice and ideas regarding the project direction that helped to tell this project's journey.

I am also deeply grateful for everything the University of Aberdeen and especially the Computing Science department has given me this year, in terms of knowledge and opportunity to grow both as a person and student.

Last but not least, I could not possibly have done this without the support of my friends and family, who cared for my well-being and supported me along the way. My greatest thank you of course, goes out to my mother who is responsible for everything I have achieved thus far, because of her love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We begin this project by giving an overview of the research and the motivation behind it, in this brief chapter. We will discuss aspects of facial recognition (FR) in artificial intelligence (AI) and adversarial attacks on AI-based recognition systems, on a higher level, to prepare the readers for the comprehensive literature overview that will follow in the next chapter. We discuss the motivation and main objectives of the project, as well as establish the critical milestones in the form of research questions by the end of the chapter.

1.1 Motivation

The continuous technological growth of the last century is undeniably reconstructing our everyday lives with each new innovation. Modern biometrics are no stranger to this phenomenon given the increasing amount of new and more sophisticated biometrics we use nowadays, especially facial recognition (FR), which albeit not as new as other biometrics, is everywhere we look now.

Evidence for the use of FR currently is shown by applications in areas such as airport security and border control in general, military operations, counter-terrorism and surveillance, even for simple everyday activities such as accessing premises, unlocking our phones or taking face filtered photographs [15, 54, 11, 40, 37, 28]. All of these applications have benefited from the rapid development of artificial intelligence technologies as well as the advancement of hardware, since not only were many of these simply unthinkable in the past but also financially and computationally costly applications. On top of all these current applications, modern FR systems are also capable of performing these incredible feats in situations such as low-light environments and across different poses [64, 42].

Due to the attention that FR is currently given, the immense availability of data and hardware advancements as well as the continuously growing interest in AI and its sub-categories like deep learning, FR specialists share an optimistic view on the field, believing that they could utilise the benefits of this technology in even more areas in the future such as retail, self-service stores, ATMs and more. Examples of popular implementations due to the influence that deep learning had on facial recognition include Deepface, Facenet and OpenFace among others [45, 57, 50, 49, 3].

Although more about the history of FR will be discussed in the next chapter, its current and future use, raises some really important questions. In reality, this technological revolution also brings about some fundamental issues with the use of this ever-changing ever-evolving technology such as concerns on model bias, privacy and security in general. This phenomena of course have been acknowledged and a call for such systems to be revised and built around robustness and not

just efficiency and precision in mind has been made.

As of late, with the global COVID-19 pandemic affecting most areas of society, people deploying FR solutions have quickly found themselves in shark-infested waters, given that a lot of them had to be reconstructed from scratch to accommodate for our new lifestyle, which focuses around the extensive use of visors, masks and other face coverings [63]. Although, adversarial attempts to influence the precision of FR systems using masks, predate the global pandemic, such examples were simply quite uncommon, and thus make this re-adjustment period harsher for FR specialists.

Cycling back to the problem at hand however, mask attacks are just one example of adversarial attempts which fall under the presentation attack category on computer vision (CV) systems. Another example are physical or digital adversarial patches [34, 10]. In this research we will not be focusing on physical attacks of any nature as is the case with presentation attacks, which attempt to modify a person's appearance to fool a system, but rather take a closer look at more digital and algorithmic adversarial attempts. Having said that, these adversarial techniques still have many different methodologies and attack strategies in regards to the time and process state at which the attack attempts to fool the system.

A comprehensive summary of all types of adversarial attacks on FR systems by Vakhshiteh et al., describes their evolution through time by presenting their contribution in literature and presents a collected general taxonomy of attacks catalogued by the scientific community [62]. Some comparisons are also made in the paper in order to better present their differences and challenges in an attempt to derive meaningful research directions for future studies in the field. A great list of adversarial attacks is also given by the Adversarial Robustness Toolbox (ART)[1] library, funded by IBM and its documentation, which offers a Python implementation of many attacks which is framework-agnostic as well as the attaching the scientific paper that proposed each attack for further inspection [41].

One could say that the increasing concern of scientists and in our case AI experts especially, would create a drive towards building such systems and technologies with overall robustness and adversarial invulnerability in mind. Nevertheless, in order to achieve that, FR systems need to be first tested against such threats in order to understand how to better fine-tune and train them to address these vulnerabilities. Despite the vast number of such attacks and the enormous amount of material available about them, this research will neither be attempting to survey every single one of these attacks nor evaluate them, as that would be impossible given the time frame given. Instead, we will focus on Deepfool, a pre-existing attack, which we will investigate and alter the algorithm further to evaluate its potency on an FR system of our making [39]. According to some, Deepfool shows more promise than some other attacks like FGSM and JSMA, due to providing smaller perturbations on images, and thus would make a good candidate adversarial attack to test on this study [62].

To achieve this, we will delve into the established literature for security in AI models and FR systems specifically, searching for documented techniques as well as technologies with their reported success and shortcomings. We will use the knowledge gathered to prepare an FR system using deep learning and tested Deepfool along with its variants and documented the results. We

[1] `https://adversarial-robustness-toolbox.readthedocs.io/en/latest/`

will model Deepfool and its variants both on a conceptual as well as programmatically intrinsic level. The first step to work upon is to set some overall goals that the project must complete to be considered a success, which will be covered shortly in the subsequent section.

1.2 Objectives

Given the motivation outlined above, we now proceed to construct the project's main objectives. The importance of this procedure is unquestionable, since it is required to qualitatively present concise goals in order to perform a valid evaluation later on, thus increasing the impact this project might have on the relevant scientific community.

Having said that, since the project involves disciplines of AI such as machine learning and deep learning for creating an FR model, we also combine common computer vision and cyber-security ideas to evaluate the adversarial part of this project. The literature investigated was instrumental in influencing this paper and giving us the ground to investigate a new form of attack on deep learning models which mainly or solely involve FR. To achieve this, we had to compartmentalise the development and research of the proposed attack to a smaller set of objectives, to better evaluate the various components of the project in our endeavours. This list is constructed and presented in order of component precedence below:

1. deep learning models and architecture
2. facial recognition systems in deep learning
3. attacks on FR systems using adversarial images

While the above list is only one way of compartmentalising the tasks and is designed in a high level manner, we continue constructing more relevant and in-depth lists of requirements in this section, which will help us understand the questions and answers that must be answered by the end of this project.

1.2.1 Research Questions

As is the case with many other scientific papers out there, this research combines disciplines of a few different computing and AI topics to create a study which is both well-defined and specific, with those topics having been introduced both in the previous section and chapter in detail. It is hence mandatory for a study such as this which focuses on a specialised field,to avoid providing a reductionist view on the relevant topics by composing a list of research questions. The following list enumerates these research questions, which are immensely important for evaluating the project:

RQ-1 Can a fairly new architecture and collection of pipeline components be used to create a facial recognition model for evaluating adversarial attacks?

RQ-2 Does adding a small random perturbation in the **Deepfool** algorithm produce, on average, more minimal perturbations than pure Deepfool?

RQ-3 Can **Deepfool's** gradient calculations be altered/improved, so that it considers more gradient points at each iteration?

1.3 Project Overview

Chapter 1: Project Introduction, Motivation, Objectivs and Research Questions.

Chapter 2: Literature overview of relevant fields and topics in Facial Recognition, AI and cybersecurity.

Chapter 3: Establishment of Requirements and experimental design.

Chapter 4: Technological overview and presentation of implemented components

Chapter 5: Testing and evaluation of the model and adversarial attacks.

Chapter 6: Discussion on limitations and future development and concluding on what was attempted and achieved in this project.

Chapter 2

Background Research and Literature Overview

We begin by performing an extensive overview of the associated literature for this project. Previous distinguished works will be discussed and analysed, in order to better understand the groundwork established by academics in the past and the drive towards more innovations today and in the future.

2.1 The history and many facets of facial biometric recognition

Facial recognition is undoubtedly one of the 21st century's most widely used biometric technologies. Yet it comes as a surprise to many, that it has been used and improved upon for over half a century, outside of science fiction depictions. In fact the earliest account of such a technology came from Bledsoe, Chan and Bisson of the Panoramic Research, Palo Alto, California, in 1964-1965, when they begun work on creating machine-based facial recognition software. Not much is known about their earliest works in the field and relevant publications are limited, given that the project was funded by an unnamed intelligence agency at the time [6, 8, 7].

Nevertheless, despite the number of technological and organisational limitations, their work introduced the first facial recognition methodology of manually computing mapped facial landmarks such as eyes, chin, mouth and others. Using the computed facial landmarks, 20 distances and the person's name were used to build the database. Together with various computer vision processes such as pose variation by inversion and known mathematical formulas, they compared these distances to perform recognition and determine the closest identities that matched these distances from the database. The system could process around 40 pictures per hour, which given the technological capabilities of that time was quite impressive [6, 8, 7].

While the torch has been passed to many other capable hands to continue their work and they did contribute for years to come in the field, some interesting progress was in fact made in the late 1980's with the use of linear algebra to assist the process of facial recognition. This breakthrough was proposed by Sirovich and Kirby in 1988 and the method quickly became known as Eigenface [53]. This method paved the way for important future work regarding feature analysis of facial landmarks, given the fact that they demonstrated the ability to accurately exemplify normalised facial images with less than 100 values/features.

Many improvements were documented in the field after that, with a very significant one being published just 3 years later (1991), by Turk and Pentland. Continuing the work of Sirovich and Kirby, they quickly became pioneers in the field of **automated** facial recognition, contrary to previously used practices which involved a lot of manual facial mapping [59, 60].

2.2 Deep facial recognition

The previously discussed short history of facial recognition was fundamental for setting the precedence in the field of facial recognition. Nevertheless, the introduction of machine learning at the time and its rapid development and academic focus in recent years, was instrumental in causing a shift in facial recognition methodologies towards what we now commonly use and know as deep facial recognition. Regardless, this is less of a divergence in focus but rather a natural evolution of methods, given that many deep facial recognition models, do still use some of the fundamental principles of FR such as facial landmark extraction and utilisation in training as well as distance comparison [29]. In this section we will talk about successful open-source FR system implementations and provide a short overview of the advancement of deep architectures for image recognition tasks.

2.2.1 Notable deep FR systems

With Tech giants like Google and Facebook turning their attention towards FR systems and providing much needed funds and experts, it was only natural that some important projects would soon see the light of day. Google's Facenet was instrumental in pushing facial recognition research further, with real life implementations sprouting to life from it such as MTCNN[1], a Python library which uses Facenet's approach for face detection [49, 65].

On the other end of FR research, Facebook assisted in creating Deepface[2] a lightweight face recognition and attribute analysis framework [50, 45, 57]. Deepface has a great collection of tools such as face recognition, verification and analysis of age, gender, emotion and race attributes.

Last but not least, it is only fair that we also mention Openface[3] a toolkit similar to DeepFace, with facial landmark detection, head pose estimation, facial action unit recognition and eye-gaze estimation capabilities [3]. In addition, Openface offers the source code for both using pre-trained models or training the models from scrath. It was originally developed by Tadas Baltrušaitis with the help of the CMU MultiComp Lab and it is still maintained under a new version [5].

2.3 Advancement of deep image recognition architectures

Deep facial recognition benefited immensely from advancements made in deep learning image recognition architectures. Despite the fact that these architectures can also be used in different areas other than computer vision, we will focus on the overall technological growth in theory and architectures, in regards to their use in image recognition problems.

Even thought the foundations of modern neural networks and by extension deep learning, relied heavily on past theoretical models such as the neocognitron by Kunihiko Fukushima, as well as the work of many of the founding fathers of AI, this paper will not perform a full history overview of technological advancements in AI [20]. Instead, we will follow the important milestones achieved from a certain point in time, were we believe the most relevant innovations were introduced and explain their importance in literature and this experiment.

[1] `https://github.com/ipazc/mtcnn#zhang2016`
[2] `https://github.com/serengil/deepface`
[3] `https://github.com/TadasBaltrusaitis/OpenFace#overall-system`

2.3.1 LeNet, Alexnet & VGGNet (VGG 16/19)

Convolutional neural networks saw an increasing attention and advancement in regards to computer vision tasks, with innovations such as Yann LeCun's early work on hand-written number and zip-code recognition, that modernised computer vision [31]. In the paper, he explained how with just minimal preprocessing, back-propagation was used to learn the kernel coefficients directly from images, and was used to create a model that achieved state-of-the-art digit recognition.

His combined published work in from his first relevant paper in 1989 up to 1998, introduced a groundbreaking 7-level deep convolutional architecture which was later popularised as the LeNet-5 model architecture. LeCun not only was instrumental in sparking the interest of exploring deeper network architectures but also became a prominent figure in certain computer vision tasks such as handwritten character recognition [31, 33, 30, 32].

Years later, a variant of LeNet-5 by Alex Krizhevsky, even though not the first to attempt a GPU implementation, it quickly popularised the idea of GPU models and became quickly known as Alexnet [27]. Its fame has yet to die out, given that it is still cited and celebrated even today as one of the most important breakthrough in CNNs for image recognition.

VGGNet is the network architecture proposed by Simonyan and Zisserman in 2014, where they introduced two architectural variants of 16 and 19 weight layers respectively [52]. VGGNet had won the ILSVRC (ImageNet Large Scale Visual Recognition Competition) in the same year the paper was published and has since been used for many real world applications, while pre-trained versions of it are freely available online.

One example of the improvements in accuracy that VGGNet offered is a research perform in 2018, on the GHIM10K and CalTech256 datasets for image recognition tasks [51]. The study showed that using the VGG19 architecture improved important metrics such as average recall, precision and F-score. Despite many breakthroughs and applications achieved by VGGNet, given the depth of its architecture poses some disadvantages compared to more modern approaches. Such disadvantages include computationally expensive demands and large number of inner parameters.

2.3.2 Recent advancements in deep architectures

Last but not least in regards to CNNs and architectures we will now talk more about modernised state-of-the-art architectures. One such architecture is that of Residual Networks (ResNet), which has been studied intensively in recent years, or that of the Inception model, currently at its 3rd version, while a hybrid between the two has been named as Inception v4 [23, 55, 56].

Other important recent architectures include the improvement of the Inception model in 2017, named Xception, created by none other than François Chollet, the chief scientist behind the Keras library and its lead maintainer , without excluding the vastly different approach of DenseNet, by Huang et al. [17, 24]. Last but not least, quite recently the introduction of EfficientNet with 7 different variants, has shown how architectures with drastically reduced internal parameters can still achieve great results, with EfficientNet-B7 managing to achieve state-of-the art top-1 and top-5 accuracy at 84.4% and 97.1% respectively [58]. The B4 variant improves the top-1 accuracy of ResNet-50 by 6.3% while being 30% smaller.

To summarise the above information on architectures, all these approaches have at least one thing in common and that is that they offer more optimised architectures that achieve better Top-1 and Top-5 accuracy with significantly less internal parameters. This is perhaps one of the most

important advantages that newer architectures offer compared to implementations of the past.

2.4 Adversarial risks in AI systems

As in all aspects of everyday life, in order to build something, either physically or digitally, is a process that should always account for security, given the need for protection from external attacks. In this regard, cybersecurity is as old as computing itself, if we were to look at its history as an real example of game theory. Nevertheless, in this section, we will redirect the focus of this large history to just one of its many subsets and focus on the security risks and concerns that surround AI and ML systems in general. Emphasis will be given to areas of great importance for this experiment such us relevant deep computer vision adversarial techniques.

2.4.1 Evasion, Poisoning & Privacy attacks

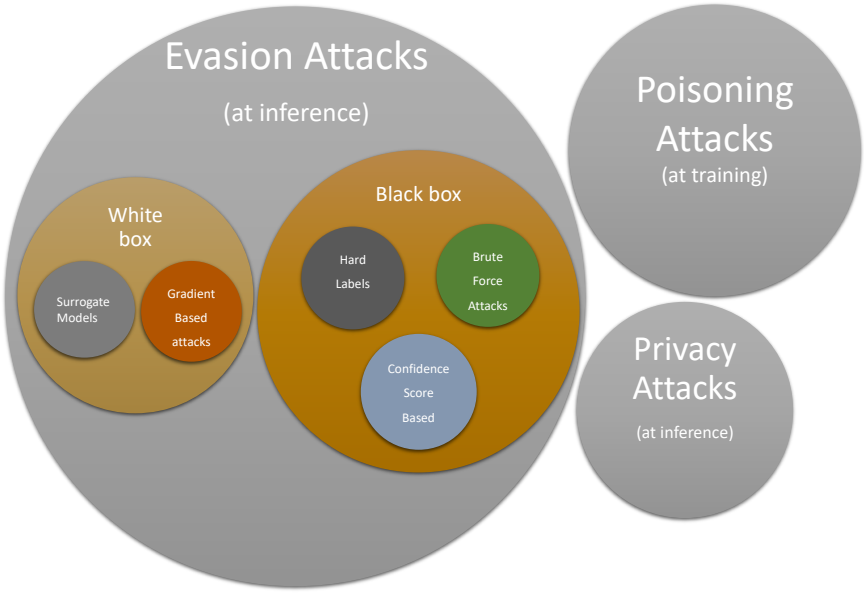

Figure 2.1: The three main types of adversarial attacks on AI systems (Evasion, Poisoning, Privacy).

Naturally, a lot of research has already been done on adversarial attacks on AI systems. Even more specific to our context was a comprehensive study by Vakhshiteh et al., in 2020, where a large number of attacks and their inherent categories are listed and explained [62]. Vakhshiteh et al. as well as many other works and material online, list at least three main categories of attacks; evasion, poisoning and privacy attacks [62, 47, 38]. Figure 2.1 has been created to illustrate these categories. Note that evasion is only presented in a larger scale because it is the main focus of this project.

As also noted in Figure 2.1, poisoning attacks occur at training, whereas evasion and privacy attacks happen at inference. This important distinction is necessary since poisoning aims to meddle with the training of the system and 'poisoning' the data it is being trained on. Despite that, to even get the chance to poison an AI system during training is very challenging. Privacy attacks on the other hand, are relatively new and only focus in getting hold of private information through their attacks during inference. This means that such attacks use this unorthodox methods because

the attackers cares less about damaging the performance of the model and more about extracting private data for their nefarious purposes.

Moving on to evasion attacks, the number of different types of attacks that fall under this category is massive. That being said, these evasion attacks are subsequently divided into more subgroups depending on their approach, such as white-box, black-box and mixed approach attacks, otherwise known as grey box. Figure 2.1, shows some approaches that fall under these two subgroups, while still maintaining an overall abstract design. White-box evasion attacks are named as such due to their approach that requires knowledge of the internal structure and components of a model to attack it, whereas a black-box approach would require no prior knowledge of a system and usually resorts to more blunt methods [38].

Black-box evasion approaches would usually try to change the output of a model or affect the predictions using very simplistic brute-force techniques such as rotating, adding noise, increasing contrast or simply perturbing the image with any common image processing technique [38]. Be that as it may, such techniques hardly ever show consistent results to perform large scale adversarial attacks especially on state-of-the-art systems. White-box approaches are undoubtedly an area worth investigating more as most advancements in improving robustness are the result of research on them. Some illustrated approaches that are listed in Figure 2.1 are surrogate models and gradient based attacks. The first refers to attacks which attempt to re-built a model, thus gaining access to its internal structure and understanding how it works, in order to launch another type of attack on it afterwards. As such, it is often followed with gradient based attacks, which are inherently sophisticated white-box attacks.

The basis of the Deepfool attack, which we will be reviewing at the end of this chapter is undoubtedly the byproduct of previous studies that theorised the concept of the gradient based attacks and implemented them for the first time [39]. Perhaps, the most acknowledged predecessor of gradient based attacks is the work of the Goodfellow, Szegedy and their colleagues which introduced the Fast Gradient Sign Method (FGSM) [22]. Moosavi-Dezfooli et al., have also used FGSM to compare its performance to Deepfool at the time, yet both of these methods have been far surpassed by more recent works such Carlini and Wagner's (C&W) attack and the Elastic Net attack, where the latter is considered one of the most used attacks today [13, 14, 16].

2.4.2 Deepfool

For the last part of this chapter we finally proceed to introduce the Deepfool attack and explain its methodology in attacking deep learning systems. Moosavi-Dezfooli et al. in 2015 introduced Deepfool as an alternative gradient based adversarial attack which focused on finding minimal norm adversarial perturbations faster and more accurately than previous attempts such as FSGM [39].

The algorithm assumes that the input image is situated between a confined region of the classifier's decision boundaries in order to proceed. Then it attempts to iteratively perturb the image by a set amount, towards the closest hyperplane of the originally predicted class. The hyperplanes are calculated by using the linear distance between the gradients of the originally predicted class label and the closest hyperplane projected label. At each iteration step, the perturbations are cumulatively added to the image until the originally predicted label is no longer the same or iterations

reach a maximum. Then the total perturbation calculated is returned. Figure 2.2 shows a reconstructed diagram of the illustration used in Deepfool's original paper, to explain the assumption of sample x_0 belonging in class 4, where $F_k = \{x : f_k(x) - f_4(x) = 0\}$ [39]. The hyperplanes are shown by the solid black lines, while the boundaries of the prediction of x_0 is marked by the green enclosed triangle inside the hyperplanes.

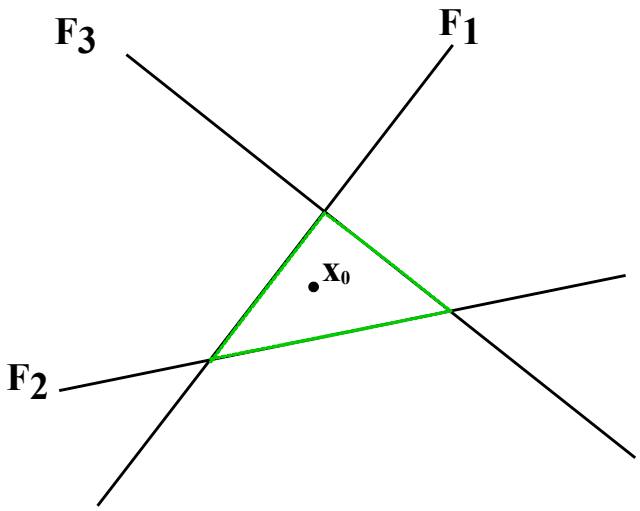

Figure 2.2: Reconstructed diagram for multi-class classifier hyperplanes by Moosavi-Dezfooli et al. [39].

Deepfool had been evaluated using its own novel adversarial robustness formula proposed in the original paper [39]. It has been reported in the paper, as well as the section above, that Deepfool outperforms certain attacks such as the Fast Gradient Sign Method (FSGM) having tested it on a number of different models at the time. In that study the test error of the models was also tested after fine-tuning with adversarial examples, with Deepfool reducing the test error of a LeNet (MNIST) classifier to 0.8% where as the same classifier was only reduced to 4.4% using adversarial images from FSGM [39]. This massive difference in decreasing test error was also noticed in the other tested models, which further proved the capabilities of Deepfool.

Despite the success of Deepfool on the reported models, years of model architecture advancements have made Deepfool quite ineffective against certain models. Similar to our objectives in this study, a research in 2020 aimed to analyse inaccuracies in the linear approximation of Deepfool and attempted to improve it in that regard [61]. The researchers behind this study introduced two different strategies to efficiently explore the gaps in Deepfool's approximation of the decision boundaries and evaluated their result on a speech command classification task. Even though this has been applied in a different domain than our case, it proves without a doubt that Deepfool can be improved to better attack and accommodate for the additional security which modern architectures are built with.

Chapter 3

Requirements and Experimental Design

This section is tasked with assembling and presenting a set of explicit and clear-cut requirements that the project must be able to meet by the end of its course to be considered a successful story. We also attempt to provide further clarity by addressing these requirements in different categories, depending on their functionality, while we delve in the methodology and experimental design shortly after.

3.1 Requirements

Now that we have a set of research questions that define the investigation goals of this project, the next step would be to establish the functional and non-functional requirements. The following subsections present each of these requirements in detail while their evaluation will follow in the Evaluation chapter.

3.1.1 Functional requirements

The research project must meet the following requirements in terms of functionality:

FR-1 Carefully construct a deep learning facial recognition pipeline on a conceptual and implementational level.

FR-2 Gather data for evaluation on the face recognition abilities of the experiment pipeline.

FR-3 Use inspiration from literature to explicitly represent a new adversarial attack on a theoretical and programmatically intrinsic level.

FR-4 Create tests on the facial recognition model to evaluate the new adversarial attack compared to other attacks and report on their performance.

3.1.2 Non-functional requirements

Usually, non-functional requirements are concerned with matters of stability, robustness and usability among others. Having said that a lot of scalability requirements can be rendered superfluous, given then impact and power of modern cloud computing solutions such as Google Colab, which provide cloud resources for the development of AI systems such as the one proposed in this paper. As such, the list below enumerates these requirements regardless:

NF-1 The research project should be able to run with relative ease on AI development environments, whether they exist in the cloud or not.

NF-2 All utilised tools and technologies such as software, libraries, frameworks and languages must be open-source or under a free license agreement and should be fully credited.

NF-3 The project should allow execution and reproduction in most widely used Operating systems, if possible and the utilised technologies allow it.

NF-4 The project must be under frequent and consistent version control and backup schemes.

3.2 Experimental Methodology

Given the set of goals and requirements above, in this section we aim to present the design decisions that will serve as the blueprint for this paper's experiments. The decisions are divided according to the sections that define the specific components of the experiment and their role in the grand scheme of the experimental pipeline. Hence we will delve into the experimental methodology, but also discuss deep learning topics, the architecture of the AI model which will be implemented as well as the dataset used among others.

In order to perform a good analysis and yield appropriate results we need to design a thorough plan of execution for our experiments. Hence, we will start by outlining the design of the components this experiment will use, and their connection to each other. A pipeline design was proposed to lay down the foundations of the necessary units of the experiment, beginning from the very first task which would be loading the data, down to the very last which would be the processing of the image or adversarial image by the AI system, for any kind of recognition task. In this linear process of building an AI model for an image recognition task, we expect that the adversarial attacks' performance will be evaluated for every adversarial attempt made iteratively.

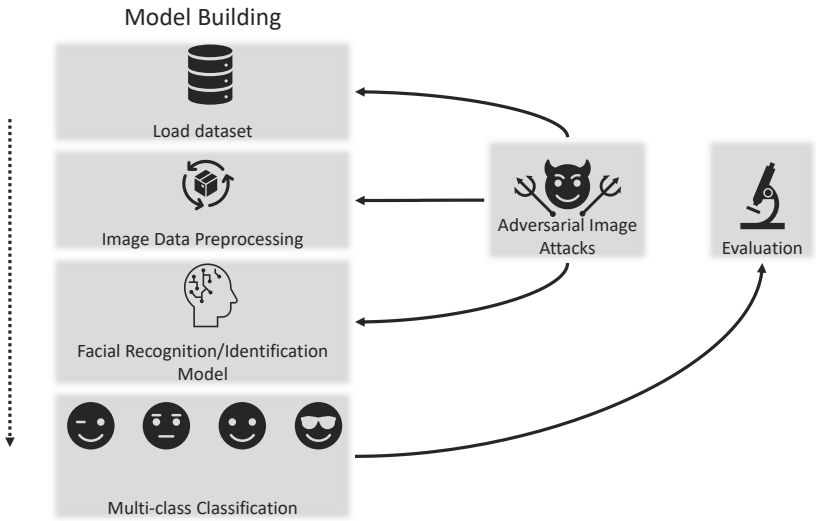

Figure 3.1: Experimental pipeline diagram.

The diagram in Figure 3.1, was created to better illustrate this experimental framework, albeit in a simple format. To clarify more, the adversarial attacks are shown as a single component regardless of how many of them will be shown in this experiment, while the training of the facial recognition/identification model is omitted in the diagram, since it will be performed only once.

This is because the diagram assumes an iterative process until all experiments are exhausted and training is done prior to that and is thus not a part of this process. The final component of the pipeline, labelled as multi-class classification refers to the output of the trained classifier given a single image, regardless of its nature (adversarial/non-adversarial).

Nowadays, as mentioned at the beginning of this document, facial recognition has been attuned for many areas and tasks. Even tough we will not attempt to recite these areas and tasks, we would like to discuss the difference in terminology that has developed over the course of this technology's history. To further elaborate, it is common nowadays to use facial recognition as a bucket term for a number of relevant recognition tasks that we can now differentiate as well as use separately for various purposes. Some like to break facial recognition down to identification and verification. Face identification would refer to the process of attributing the correct identity to a given recognised face. On the other hand, face verification would refer to the process by which we can compare faces to determine whether the two faces are identical, or belong to the same identity. Having said that, it should now be easy to see how these two processes are more connected than separate in theory, since to identify a person, we usually have to verify a given face over a database of faces matched with an identity.

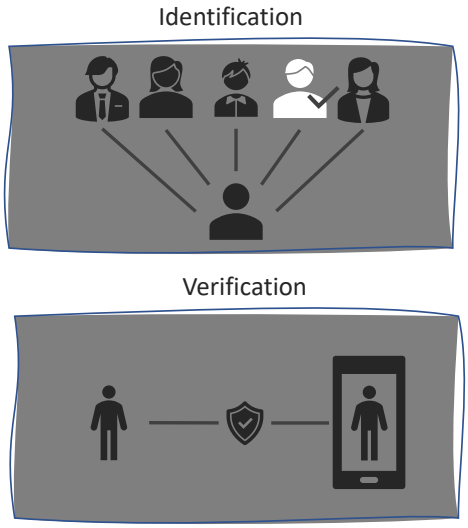

Figure 3.2: Diagram created to illustrate the difference of Identification and Verification. Identification demonstrates a 1-to-n (many) matching whereas verification performs a 1-to-1 check.

3.3 Dataset

One of the most difficult tasks in creating an AI system is perhaps the data collection part. Choosing a suitable dataset which has good quality of data and has sufficient quantities for the problem at hand is undoubtedly a struggle sometimes. Having said that, it was highly necessary to conduct an investigation on what datasets could be used for this experiment and determine the winning candidate.

According to Vakhshiteh et al., LFW, CASIA-WebFace, MegaFace, VGGFace2, and CelebA

are the most widely used image classification datasets to evaluate adversarial attacks on FR systems [62]. With this in mind, a number of face recognition datasets were considered for the tasks ahead, inlcuding CelebA[1], VGGFace[2], VGGFace2[3] and LFW[4] among others[5] [12, 25, 35, 45]. All of the aforementioned datasets have been previously used in well established scientific experiments regarding facial recognition, while some of them, such as LFW and VGGFace are considered a benchmark standard for evaluating architectures, algorithms and more.

The initial thought-process of the project leaned towards using CelebA dataset, given that it contained more than 200k celebrity images. Furthermore, the group behind the CelebA dataset had also released an anti-spoofing dataset for facial recognition model. Despite that, the dataset was significantly smaller in quantity, while a large portion of its facial images could be deemed inappropriate for this type of research, as they are primarily used for presentation attacks with static images, masks or printed faces. LFW on the other hand, might have been a smaller dataset choice, since it contained around 13k images, yet it is known for its good data quality.

Finally, the decision to use the VGGFace2 dataset, the successor of the first VGGFace was eventually made. The reason for this is that VGGFace2 boasted an even larger and more diverse collection of faces than most datasets investigated, amassing an approximated total of 3 million images. In spite of that, while experimenting with the data, it is clear that most of it will not be used, as it would be nearly impossible to scale such large quantities of images in memory on a non-AI-ready machine or even with the help of cloud and distributed AI services. Nevertheless, it provides us with a great advantage of choosing from a plethora of good quality images from 9131 subjects.

Be that as it may, we will not be attempting to use such large number of identities either. We cannot replicate large scale experiments such as benchmarking architectures on the Imagenet dataset which has enormous amounts of classification classes. We therefore concluded to the use of 6 main classification classes from identities of the dataset, chosen for their diversity of facial images and large number of samples. In Figure 3.3 you will find a graph showing these 6 identities and their number of samples used. It is worth noting however, that these identities were chosen for the high number of available images as well as overall quality of samples. Figure 3.4 shows the sample numbers by gender.

[1] http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
[2] https://www.robots.ox.ac.uk/~vgg/data/vgg_face/
[3] https://github.com/ox-vgg/vgg_face2
[4] http://vis-www.cs.umass.edu/lfw/#explore
[5] http://vision.ucsd.edu/content/extended-yale-face-database-b-b

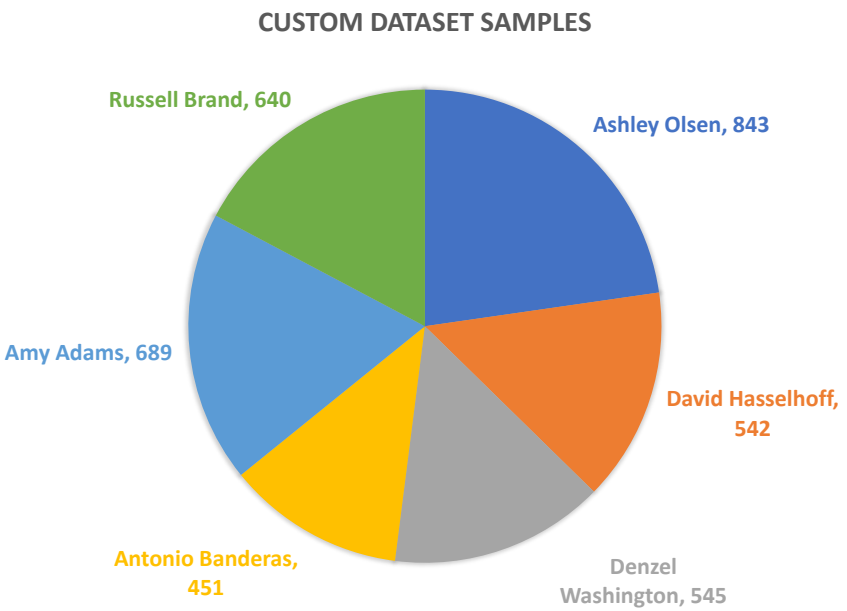

Figure 3.3: The custom dataset build from the original VGGFace2 dataset. The chart shows the total samples of each identity.

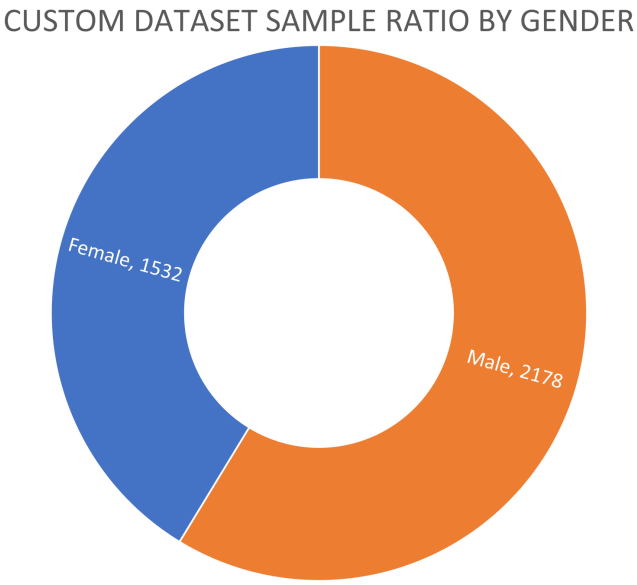

Figure 3.4: Sample ratio of identities by gender.

3.3.1 Ethical Considerations

Even before the commencing of the project in terms of development, it was necessary to file for ethical approval of this scientific work. Given the nature of this project, it was of great importance to consult the ethics board in regards to the data which will be used for our experiments. Since the dataset used is open-source, and participants were not needed, all other aspects of evaluating our work involved few risks. Hence, ethical concerns were limited and ethical approval was granted rather quickly.

3.4 Image Prepossessing

Standard Image recognition practices in AI models usually demand for the input data to be prepossessed prior to being fed to the neural network. In our case, preprocessing is quite simple and yet it must be deconstructed and examined in this section to establish its importance. The first step that we take in preprocessing is **extracting faces** since a major advantage of deep FR systems is that we do not need the whole image to perform image recognition. After successfully extracting the faces we can begin performing necessary preprocessing techniques, such as **re-scaling** the image to an adequate size which will not be too computationally demanding.

The next step is to normalise the images on a scale between -1 and 1. Image **normalisation** will ensure that all image values are normalised in a specific and more easily recognisable range, which can be arbitrary for human perception but does in fact show varied results in deep learning. Finally depending on the model architecture that we use, we might have to include additional preprocessing techniques, and hence will be further discussed in **Chapter 4**, where we cover implementation.

3.5 Architecture

To better address the problem at hand, we need to be able to understand the difficulties that lie therein. As such, given that facial as well as image recognition in general are computer vision problems, which oftentimes require very extensive and rigorous tools to handle them. As previous studies have very well established by now the difficulty and intensity of machine learning problems involving vision, they have also demonstrated the effectiveness of deeper architectures involving convolutions, as was the case with ALEXNET and VGG16/VGG19 and others that followed [27, 52]. Hence, in the subsections that follow, we will talk about convolutional neural networks and the chosen architecture that could be used for our experiment.

3.5.1 Convolutional Neural Networks (CNN)

In the following diagram in Figure 3.5, we show a common example of what a convolutional neural network classifier and its components look like in terms of architecture. The combined use and order of convolutional and pooling layers is a common trend in regular CNN architectures, yet recent advancements have introduced even more ingenious architectural patterns that show even better results. The emphasis however, should go towards the two main units of CNN architectures which is of course the feature extraction and classification.

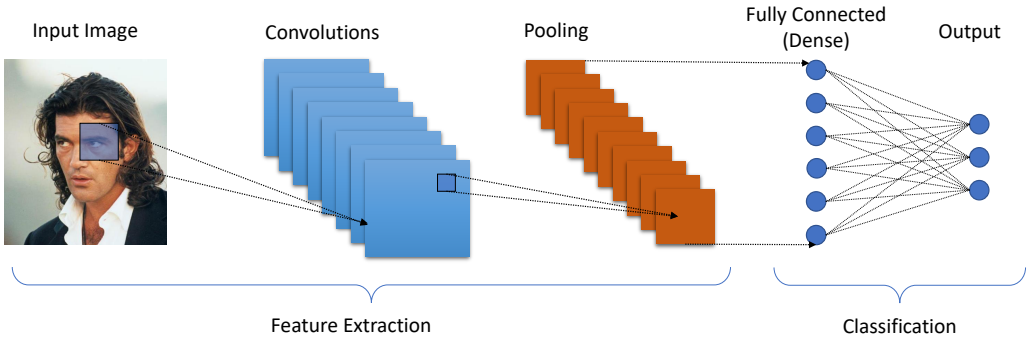

Figure 3.5: A diagram created for this project to describe the common architectural components in CNN classifiers, such as convolutions, pooling and the fully connected last layers. The input image used is part of the VGGFace2 dataset, and the structure of the diagram was influenced by similar diagrams from online sources [36, 48, 2].

As evident from the diagram as well, it is a common practice to use convolutions with specified filters and strides to scan the image according to the dimensions given, such as (*Batch ,Width, Height, Channels) or reversed, depending on the model and methodology. The output of the Convolutional layer will be a feature map which will then be passed to a pooling layer to perform down-sampling. There are different kinds of down-sampling operations for different purposes and scenarios, yet their main function remains the same.

Finally, the fully connected layers are used for the output of the model, given that they flatten the previous layer's output in order to reduce the output according to the specified neurons. If we are to use our case from the proposed methodology, an example fully connected output layer would have 6 specified neurons so that he model outputs an array of 6 probability values, one for each class. Based on this architectural framework illustrated in the diagram as well as more recent alternative iterations, we must account for possible solutions for the tasks that we must perform.

3.5.2 Transfer Learning

We have by now talked extensively about CNNs and CNN architectures for FR and computer vision in general. Even though we can utilise the great benefits of well-established deep CNN architectures, there is a broader approach that elevates this strategy even more. Not only can transfer learning be used to train a model with a decent established deep architecture, but it can also speed up the process and help reduce overfitting when data is limited [43]. This is achieved by what is called **knowledge transfer**, which utilises a pre-trained model's weights to improve feature extraction in a new model. In order to accomplish that, it is necessary for the pre-trained model to trim its old output layer and create a top with a new output layer, featuring the revised, correct amount of neurons required for the new task. It is also important to mention that in order to train the new model for the task, we need to train the model with all layers frozen except the

new top, since we do not want to change the already good weights that we transferred.

Nevertheless, what was described above, is quite often just the first phase of a successful application of transfer learning. According to Géron, training with frozen layers should continue until approximately 75% to 85% validation accuracy is achieved, while what follows after this, is the next and final phase of transfer learning, commonly known as **fine-tuning** [21]. In this final phase, we attempt to unfreeze all layers in order to holistically train the model again. By using a very small learning rate and training the model for an arbitrary number of epochs relevant to the domain problem, the model's weights are gradually tweaked and validation accuracy can further increase, hence the name fine-tuning [21]. In Figure 3.6, we demonstrate the process of transfer learning on an abstract level, where fine tuning is omitted for clarity. As you may notice, we mention faces in the second domain's output to better explain how this technique can be applied to our experiment.

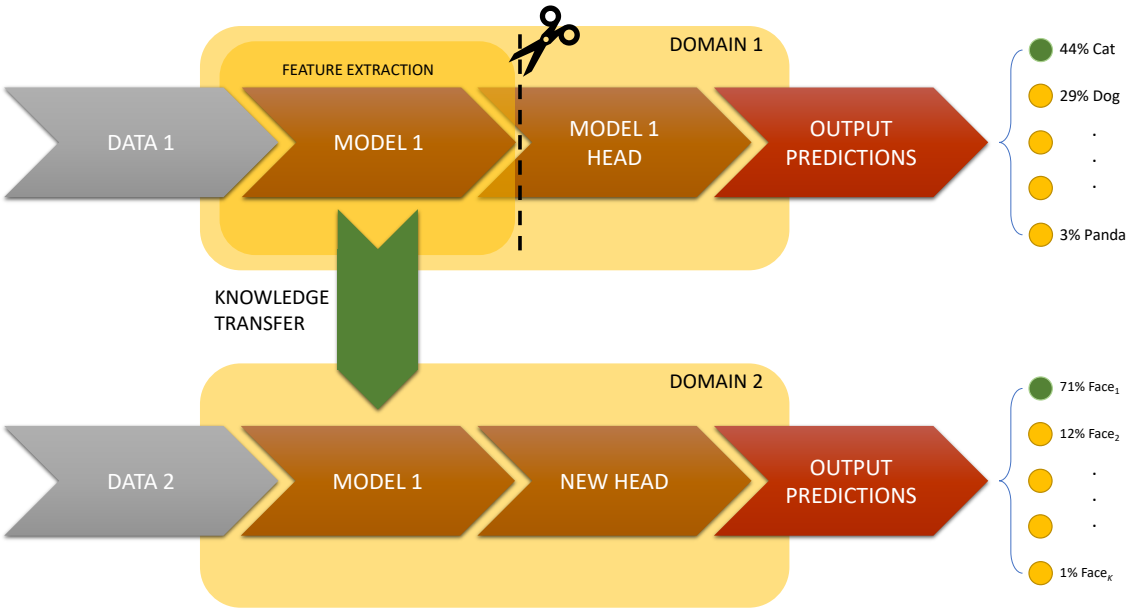

Figure 3.6: A diagram created to explain the process of transfer learning. It shows how a model applied for Domain 1 can be trimmed at the top and used in Domain 2 with a new output head for a different classification problem.

The list of studies on transfer learning for image classification is quite impressive, and could perhaps be the subject of an overall technological evaluation in the future. Nevertheless, in this section we are merely pointing out a few examples of this phenomenon, in order to demonstrate the design choice of using transfer learning in this experiment. That being said, some implementations of transfer learning in facial recognition did in fact show that models with knowledge transferred perform well given the lack of data in their problem domain. One such case is a study by Hussain et al., in which they tested an Inception-v3 transfer learning model on a CIFAR dataset [6] , as well as a the CalTech [7] facial recognition dataset [26]. Another study by Shaha et al., used the VGG19 architecture and fine-tuned a model on the CalTech and GHIM10K datasets [51].

[6] https://www.cs.toronto.edu/%7ekriz/cifar.html
[7] http://www.vision.caltech.edu/Image_Datasets/Caltech_10K_WebFaces/

In terms of deployment practices, we should however mention that a lot ML libraries such as Keras, package transfer learning models with different parameter options for weights [18]. The options include **no weight** parameters for an untrained network, a specification to the **path of a trained network's weights**, or finally the option that we want to focus on and the most common choice, the use of weights trained on the **ImageNet dataset** [19]. Imagenet is a hierarchical image database, which was created to join previous known industry standard databases and became a popular choice for preparing transfer learning models and running benchmarks on architectures and models. It was created with 12 sub-trees of categories and 3.2 million high resolution annotated images, which for the time was very impressive.

3.6 Adversarial Attacks (Deepfool variants)

Having introduced adversarial attacks in the literature overview, we are now more versed in what attacks we can attempt on the model. Since we also have access to the internal mechanisms of the model, it is possible for us to launch some **white-box evasion** adversarial attacks on the model. Be that as it may, there are vast amounts of documented attack types, hence the focus of this experiment heavily lies on exploring just one of them. In our case we will put Deepfool under the microscope and also experiment on it, in order to test possible augmentations on the original idea [39].

In order to develop the adversarial attacks, a lot of online resources[8], repositories[9,10,11] and tutorials will be utilised in combination with scientific papers such as the original paper of the Deepfool algorithm [44, 4, 39]. A fair amount of code will need to be adjusted to re-implement pure Deepfool, given the fact that a fair share of code examples were using either outdated or alternative libraries to the ones he have nowadays and will attempt to use in this experiment. The online resources and tutorials read in order to achieve this are credited fully in the footnotes and also cited in this section as well as **chapter 4** where implementation is covered.

[8] https://blog.floydhub.com/introduction-to-adversarial-machine-learning/#deepfool
[9] https://github.com/MyRespect/AdversarialAttack/blob/master/deepfool_tf2/deepfool_tf.py
[10] https://github.com/LTS4/DeepFool
[11] https://github.com/iArunava/scratchai

Algorithm 1: DeepFool for multi-class case

input: Image x, classifier f

output: Perturbation $\hat{r}$

Initialise $x_0 \leftarrow x, i \leftarrow 0$.

while $\hat{k}(x_i) = \hat{k}(x_0)$ **do**

  for $k \neq \hat{k}(x_0)$ **do**

    $w'_k \leftarrow \nabla f_k(x_i) - \nabla f_{\hat{k}(x_0)}(x_i)$

    $f'_k \leftarrow \nabla f_k(x_i) - f_{\hat{k}(x_0)}(x_i)$

  end

  $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(x_0)} \frac{|f'_k|}{\|w'_k\|_2}$

  $r_i \leftarrow \frac{|f'_{\hat{l}}|}{\|w'_{\hat{l}}\|_2^2} w'_{\hat{l}}$

  $x_{i+1} \leftarrow x_i + r_i$

  $i \leftarrow i + 1$

end

return $\hat{r} = \sum_i r_i$

Figure 3.7: The Gradient-based Deepfool Algorithm [39].

In Figure 3.7, you can see the Deepfool algorithm for multi-class classification problems. Some more adjustments to the original algorithm are made for multi-class problems which are explained in the original paper. We will therefore refer back to them again in the next chapter when we implement the adversarial attack and its variations.

3.6.1 Spiked Deepfool

As the research questions dictate, we should test whether whether a Deepfool variant with added small random perturbations can produce less total perturbations and measure it's efficacy compared to the original attack. To do so we propose the following addition to the original algorithm, where it takes place after the addition of the iterative perturbation to the minimum total perturbation. That means that this random perturbation has a chance of occurring at every iteration of the algorithm, dictated by the probability of it's occurrence which is not static. The code for the added random perturbation is shown in Figure 3.8. It should be noted that in this variant, the return statement of this addition to the original algorithm, also changes the return statement at the end of the original pseudo-code where the minimum total perturbation is returned. In simple terms, the algorithm shown in 3.8 is just an addition at the end of each iteration of the original Deepfool algorithm.

Algorithm 2: Spiked Perturbation

input: Total Perturbation $\hat{r}$, Iteration Perturbation r_i

output: Perturbation $\hat{r}$

Initialise $\theta \leftarrow 0$. where $0 < \theta \leq i$ and $i = 50$

if $i \bmod \theta = 0$

then

  $r_j \leftarrow (r_i)(p)$

end

return $\hat{r} = \sum_{i,j}(r_i),(r_j)$

Figure 3.8: Spiked Perturbation for Spiked Deepfool variant, where θ is a random arbitrary integer, i is the max iteration of the algorithm and p is a fixed multiplicative factor.

3.6.2 Quadruple gradient acceleration (QGA) Deepfool

Finally the most challenging and perhaps important variant in this experiment will be the Quadruple gradient acceleration (QGA) Deepfool. This variant differs from the original Deepfool and the previous variant Spiked Deepfool, in that it utilises gradients even more for some of its calculations. To elaborate further, this Deepfool variant, contrary to the original, calculates **4 additional gradient** points at each iteration. At the point where the original Deepfool attack would store the minimum difference between gradients, we instead compare the difference between the 4 additional gradient points with the original gradients, and store only the one with the greatest gradient difference. The greatest gradient difference would allow for a *simulated jump* when it is used for calculating that iteration's perturbation. Figure 3.9 shows the quadruple gradient acceleration addition to the original Deepfool algorithm.

Algorithm 3: Quadruple Gradient Acceleration.

input: Perturbated Image x_i, Current Gradients $\nabla f_k(x_i)$, Original Gradients $\nabla f_{\hat{k}(x_0)}(x_i)$

output: Adjusted Gradient Difference w'_k

Initialise $A \leftarrow \{h, h', v, v'\}$

$w'_k \leftarrow \nabla f_k(x_i) - \nabla f_{\hat{k}(x_0)}(x_i)$

for a ***in*** A **do**

  $aw'_k \leftarrow (\nabla f_k(x_i) + a) - (\nabla f_{\hat{k}(x_0)}(x_i))$

  if $w'_k < aw'_k$ **then**

    $w'_k \leftarrow aw'_k$

  end

end

return w'_k

Figure 3.9: QGA adjustment for QGA Deepfool variant, to adjust gradient difference calculation.

Chapter 4

Implementation

The literature overview as well as the requirements and design that preceded this chapter should now provide an understanding of what we aim to achieve. Continuing from there, in this chapter we will focus on presenting the major implementational milestones in a modular manner and establishing the details of the modelled information. Important decisions regarding the implementation structure and choices will also be discussed and explained further while a technological overview will come first.

Lastly, the process of implementation will be analysed at the end of the chapter, demonstrating the development of modelled components from purely theoretical blueprints to realised and fully functional code.

4.1 Technological overview

To successfully implement the components proposed in previous chapters we have to first introduce the candidate as well as chosen technologies that we will utilise in this research project. To make these decisions, a combination of personal experience and online research on relevant resources was used, to accumulate a list of possible choices.

4.1.1 Machine learning and utility libraries

Given the knowledge and skills acquired during this academic year, the main candidate libraries used are without a doubt Tensorflow, Keras and Scikit-learn [1, 18, 46]. There have been quite a few new additions in machine learning libraries as of late, with many of them in now different programming languages such as Java and JavaScript. Nevertheless, a decision to mainly use the three aforementioned libraries was taken due to the amount of exposure and practice performed on them during the course of this year's studies. Moreover, these libraries are all python libraries which are often used together with great compatibility, which renders the possibility of technological inconsistencies very low. With machine learning being the centre of attention in the field of AI for years now, these python libraries are absolutely dominating most online tutorials, guides and resources.

- Tensorflow[1]
- Keras[2]

[1] `https://www.tensorflow.org/`
[2] `https://keras.io/`

- Scikit-learn[3]
- Opencv-python[4]
- numpy[5]
- Matplotlib[6]
- Seaborn [7]

4.1.2 Working environment & IDE

Given the nature of this experiment, certain development tools had to be used while coding such as an IDE and an AI development environment. To achieve this, it was not necessary to do any extensive research, and instead used knowledge and personal experience acquired throughout this year's courses.

Having said that, choosing an IDE was not particularly difficult since available options were quite standard. Given the fact that most AI development is performed in interactive python kernels such as python notebooks, the options for IDE where boiled down to a handful. Pycharm[8] is a great example of a powerful and extensive IDE with built-in working environment creation and configuration. Nevertheless, python notebooks are not supported in the free Community Edition, but are instead offered in the Pro version. Due to the price margin, this option was quickly dropped.

One the most popular IDE options nowadays is Microsoft's VS Code[9]. It is used by many developers in all platforms and it is easy to see why. It offers an impressive array of extensions for any type of developer, support for specific libraries, automated library import, code suggestions, error highlighting, code auto-completion and many more. Contrary to Pycharm, VS Code does provide support for python notebooks through extensions. Fortunately, I have worked with VS CODE in the past and based on my personal experience with using it as well as its many benefits, VS Code was used for a good part of this project's development, as it provided me with all the utilities that the other alternatives did not. Undeniably, these benefits were all instrumental in increasing my efficiency and making tedious tasks easier to handle.

As previously mentioned, development requires the use of python notebooks. Hence, that constitutes the need to use Jupyter[10]. Jupyter offers an interactive python kernel to execute code in isolated cells in a prepared virtual environment. Jupyter notebooks offer great tools and are known for providing a more dynamic environment for AI developers. Having said that, I have used both regular Jupyter notebooks as well as Jupyter Lab, which is a relatively new service, providing a refined and sophisticated IDE for python notebooks. It can also be be improved with extensions, despite the fact that support for extensions is still quite limited in its development phase.

Although, the above IDEs where used for the majority of the development of this experiment, I have also utilised Google Colab, which is an excellent cloud service for AI developers, given the

[3] https://scikit-learn.org/stable/
[4] https://opencv.org/
[5] https://numpy.org/
[6] https://matplotlib.org/
[7] https://seaborn.pydata.org/
[8] https://www.jetbrains.com/pycharm/
[9] https://code.visualstudio.com/
[10] https://jupyter.org/

fact that a good portion of those services is provided for free. Even though the code was not regularly executed in Colab, since my working machine has better specifications for training and evaluating AI models, Colab creates a fully functional working environment for AI given that many required libraries are already installed and ready to use, with much attention given towards robustness. Library incompatibilities and other dependency clashes are very rare in Colab and it is one of the main reasons why it was used in this project. Moreover, some very basic features of Google Colab, such as syntax highlighting, code collapse and in general a very simple but polished look, tremendously help during development since they improve readability and writeability.

4.1.3 Source Control/ Version Control Systems

Another important issue regarding employed technologies in this project is source control. Even though it is not a mandatory requirement to use source control for this experiment, it has in fact become an unspoken truth for any developer. The reasons behind this are quite transparent, since it is important to have back ups of different versions of the code developers work on. Having various structured and documented versions to revert back to in case something goes wrong for any reason, as well as creating and merging different branches are all very powerful functions that Version Control Systems can offer.

Following that mindset, Github's desktop application[11] was utilised to source control the development of this experiments code in a private repository. This decision proved to be invaluable in the first stages of development, given the natural need to try out multiple different technologies before development starts. The desktop application was instrumental in providing essential comments for documentation whenever an important change or a new component was made, while its intuitive user interface made GIT requests such as commit, push, pull, revert and many others feel effortless.

4.2 Model source code overview

As we have already established in the experimental design section, to evaluate the adversarial attacks on a facial recognition system, we are required to implement one such system. The quality of the system is undoubtedly less than state-of-the-art alternatives, yet the decision to implement an FR system is mainly attributed to the need of explaining the underlying processes within facial recognition while we perform the evaluation and how these processes are important as well as how can they be exploited. Nevertheless, the quality is sufficient for the evaluation that we aim to perform, while any limitations that should arise will be reported and worked upon in the future.

In order to streamline the process of building and testing AI models as much as possible, a number of utility functions have been designed to help with the problem at hand. Nevertheless, the process is intrinsically linear since we have created orderly sections of code blocks in a well documented python notebook. Below is a list which is also present in the code and serves the purpose of explaining the processes that will be performed to load the data and preprocess it in order. In this section we will also attach the corresponding code section in the notebook for easy referral.

1. Extract faces (**Section 1.2.1 - 1.2.2**)

[11] `https://desktop.github.com/`

2. Replace images with the extracted faces (**Section 1.2.2**)
3. Rescale the images to a smaller size (**Section 1.2.1 - 1.2.2**)
4. Clean dataset of any non-recognised faces since it will return None (**Section 1.2.3**)
5. Saving/Loading the train and test sets after preprocessing as numpy objects (**Section 1.2.3 - 1.2.4**)
6. Encode class labels to one hot arrays for the classifier to use multi-class classification with categorical cross entropy (**Section 1.2.4**)
7. Split the dataset into train and test arrays (**Section 1.2.5 - 1.2.6**)
8. Cast to Tensorflow dataset and create batch organised samples (**Section 1.2.7**)

4.2.1 Face extraction and image rescaling

Let us now present the process demonstrated by the list above by providing further insight for each step. As previously mentioned in the previous chapter, a whole face recognition dataset would not be utilised for this project, given the immense amount of computational and hardware requirements compared to the machine that development took place on. As also established in the last chapter, 6 identities were chosen to build a smaller dataset to run the experiments. This process of loading the images in memory happens at the same time as face extraction, hence its position as the first three processes in the list. The reason behind this is that it is quite redundant and not memory efficient to first load the entire images for all identities.

What was performed instead was iterating over the directories of identities, picking each image and performing face extraction on the spot, which will be then saved in a numpy array. Face extraction was performed using **OpenCV** from a tutorial[12] which was used to build the face extraction function in **Section 1.2.1** [9]. We also rescale the images to fit a certain fixed size for our model, which in this experiment was set to (160, 160, 3), meaning 160 pixels for width and height and 3 channels for colours in RGB format. Following this methodology of performing this essential pre-processing techniques on the spot, proved to be more efficient than performing these operations repeatedly on a dataset of entirely loaded images. Using only the extracted faces also helps in reducing features, by collecting only meaningful features regarding facial recognition.

4.2.2 Data cleansing and save/load functionalities

After successfully performing the first 3 operations we must then account for instances where a face was not recognised by the face extractor. When that happens, the functions return **None**, instead of a numpy array of shape (160,160,3). We therefore remove instances of **None** type from the dataset, along with their labels, to clean the dataset from empty values. The sets are then saved separately as numpy objects which can be loaded with ease to improve coding practices.

4.2.3 Categorical data conversion

Once we have built our image and label arrays we must then encode our class labels to categorical format. A mixture of Scikit-learn and Tensorflow material was used to achieve that [46, 1]. A

[12] `https://www.digitalocean.com/community/tutorials/how-to-detect-and-extract-faces-from-an-image-with-opencv-and-python`

LabelEncoder object from Scikit-learn was used on our dataset to encode all labels in numerical format, meaning that they are converted to numbers of 1-6 since we have six classes. Then the dataset is passed through the *tf.keras.utils.to_categorical(...)* function to convert all labels to one hot arrays. This step is required because of the loss and activation functions that we are going to use later on in our model.

4.2.4 Splitting the dataset into arrays

The next step is to split the whole dataset into the train and test sets respectively. For this procedure the most common method to achieve this result is to split the dataset with Scikit-learn's *train_test_split(...)* function [46]. A split of 0.3 was used, which corresponds to 70% of the whole dataset for training and 30% for testing. We also use a *random_state* parameter to signify that the split should not be random every time this code block is executed.

4.2.5 Final preprocessing and batch preparations

As a final pre-processing method, a number of effective measures and techniques are used to make training faster and more optimised. A function called *preprocess(image,label)* is re-defined from Tensorflow's documentation to better suit this classification problem. The function first casts the image to a Tensorflow float object and then applies the corresponding preprocessing which matches the model architecture used. Tensorflow provides these built-in preprocessing functions which can be modified in the way it was also modified for this study.

The function is designed to be used on tuples of **(image, label)**. Regardless, this is not an issue as both datasets are cast to Tensorflow datasets and their own *map(preprocess)* method is used to perform it iteratively on the whole dataset. Prior to that the dataset is shuffled and then prepared in batches of a specified size of 32. The batch size is very important to decide with careful testing, however our unofficial and undocumented tests agree with Géron, who quotes LeCunn on why batch sizes of 32 and less are preferred [21]. Lastly, we use the *prefetch(...)*[13] method call to enable the dataset to be fetched from memory faster. This makes a significant difference while training on the dataset.

4.2.6 Transfer learning

Following the pipeline design, after loading the data and applying preprocessing, the next step is to build and train the FR models. As outlined in the Design and requirements chapter, we will be utilising **transfer learning** to build our models with pre-trained weights. To do so, a number of online tutorials[14,15,16] and other resources were gathered and used in conjunction to build this part of the pipeline [21].

Before the implementation of transfer learning models is demonstrated shortly, it is important to explain the rationale behind the models which will be used for this experiment. After careful consideration and resources online[17] that report on modern architecture's performance in regards to top-1 and top-5 accuracy, the architectures that were chosen were Xception, Inception V3 and EfficientNet-B0. All three architectures are especially good in terms of performance with Xception

[13] https://www.tensorflow.org/guide/data_performance
[14] https://codelabs.developers.google.com/codelabs/keras-flowers-transfer-learning
[15] https://keras.io/guides/transfer_learning/
[16] https://www.tensorflow.org/tutorials/images/transfer_learning
[17] https://keras.io/api/applications/

taking the lead, followed by Inception v3 and EfficientNet-B0 being the last. A case could be made as to why would EfficientNet-B0 and not any other variant from 1-7 would be used. The reason behind this is that even though higher EfficientNet variants seems to perform even better than the Xception and Inception v3, the number of parameters is around the same or more. Deeper architectures however, aren't always preferred. In this case the B0 variant was used because of its outstandingly low number of parameters (≈ 0.25 of Xception's size), which gives it an edge in speed during evaluation. After all, the aim of the study is to conduct an experiment of adversarial robustness and test Deepfool variants, hence using the absolute best model architecture is not the main objective.

Going back to the implementation of a transferable model, it can be called with the method shown below:

```
base_model = tf.keras.applications.Xception(
    weights="imagenet",
    input_shape=(160, 160, 3),
    include_top=False)
```

As you can see an instance of an Xception model from Keras applications is called with certain parameters, one of which is the weight initialisation which is set to ImageNet weights. This is desirable since we need these weights to transfer knowledge of feature extraction to the new model. We also specify the input shape and specify the exclusion of the top of the model, which uses the old classifier. This way we can rebuild the model with a new head for the new classification problem. Once the model is created with the new head we can display its structure as shown in Figure 4.1.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 160, 160, 3)       0
_________________________________________________________________
xception (Functional)        (None, 5, 5, 2048)        20861480
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 6)                 12294
=================================================================
Total params: 20,873,774
Trainable params: 12,294
Non-trainable params: 20,861,480
_________________________________________________________________
```

Figure 4.1: Example model summary of a rebuilt Xception model.

This layer structure shown in Figure 4.1 was also used for the other two transferred architectures. The Figure also shows how the model receives an input of size **(batch, 160, 160, 3)** and produces an output of **(batch, 6)** since we expect an array of six prediction values, one for each of the 6 classes. Just after the input is passed to the model and before it is passed to the main body of the transfered layers, it is first going though a sequential mini block of **data augmentation**. This block is essential for a number of reasons such as reducing overfitting by introducing different angles and rotations to images and thus synthesising more varying training samples. The transferred layers then pass their output to some newly defined global average pooling and dropout layers which also help with overfitting.

4.2.7 Initial training

Since a model is now built, the next step is to train the model and fine-tune it. We first make sure to **freeze** the layers and weights of the transferred model while we initially train the new model top. The models have been tested with quite a few hyper-parameters and configurations and the best ones are reported in this section. All models have been trained with a **Categorical Cross Entropy loss** which requires one-hot arrays for multi-class classification problems. Note that we use this loss from logits, since Deepfool works on logits and not on post-activation probabilities. Moreover, we compile our model with an Adam optimiser and an initial learning rate of 0.02 for 50 epochs. From previous attempts it has been observed that Adam was performing better than other optimisers such as Nadam and stochastic gradient decent.

Despite the above details on hyper-parameters, the implementation of additional safety-net callbacks was deemed necessary. Namely, to better optimise the learning rate during training, Exponential learning decay[18] was used at first, but was quickly dropped due to ineffective results. A better suited callback[19] was used in its place which reduces the learning rate only if a **plateau** is reached on a **monitored metric**. Hence, we used this callback to monitor **validation loss** with a **patience** of 4 epochs of not improving and a factor of 0.25 for reducing the learning rate.

4.2.8 Fine-tuning

Following Géron's advice on training the head of the model, once training had reached about 70-80%, the training was stopped as it will plateau regardless [21]. In order to fine-tune the model, we must now **unfreeze** the weights of the transferred model to fine-tune its feature extraction capabilities to our classification problem. The model is **recompiled** with the same loss and metrics as with its initial training phase, however we now use a significantly smaller learning rate of 0.0001 with an Adam optimiser for 150 epochs. This time, there is no need to use a learning rate scheduler, reduction or decay of any kind since the learning rate is already very small. Instead we use an early stop callback[20] which can stop training and save the best epoch if the monitored metrics does not improve over 30 epochs. Following this approach allows for reduced training time when the model has already converged to a good performance, which is the case with all three models, since they converge very fast. Last but not least, for both the initial and fine-tuning training of the model, we use a callback[21] to save model checkpoints of each epoch. This allows us to also load[22] the model

[18] `https://keras.io/api/optimizers/learning_rate_schedules/exponential_decay/`
[19] `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau`
[20] `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping#args`
[21] `https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint`
[22] `https://www.tensorflow.org/tutorials/keras/save_and_load`

from file if there is a need to do so.

4.2.9 Model evaluation functions and code blocks

The following functions and code blocks have also been implemented to evaluate the model. The size of each of these functions does not allow for a thorough overview of their inner processes and functionality, however the list below should provide a good enough description for each of them. The section that corresponds to each of these functions in the source code notebook is also included for the readers convenience.

1. Training/Validation Accuracy/Loss Plot (Initial Training) (**Section 1.3.4**)
2. Training/Validation Accuracy/Loss Plot (Fine-Tuning) (**Section 1.3.4**)
3. Preparing/Recasting test set (**Section 2.2.0**)
4. Display Predictions (**Section 2.3.0**)
5. Decode Predictions (**Section 2.3.0**)
6. Gather Predictions (**Section 2.4.1**)
7. Plot Confusion Matrix (**Section 2.4.2**)
8. Classification Report (**Section 2.4.3**)
9. Gather Miss-classified samples (**Section 2.4.4**)
10. Random Single Predictions (**Section 2.5.0**)

4.3 Adversarial attacks code overview

As previously mentioned in Chapter 3, several online repositories and guides have been used to implement the Deepfool attack variants. A lot of difficulties have been observed in implementing the attacks on the three models, given their overall inherent robustness but also the numerous API bugs found in the libraries used. In the subsections that follow we provide an overview of the implementation code for the three variants as well as the functions used to evaluate attacks and adversarial robustness of the models. A complete walk-through of the code will not be performed to maintain sizeable chapters. Instead, a thorough dive into the code will be made in the Instruction Manual in **Appendix B**.

The list that follows, enumerates the important code blocks regarding adversarial attacks with their respective section in the notebook included. Each of the attacks has been programmed as an object oriented class which can be used to create an instance and perform a single attack with displayed predictions and plotted images using the **Single Attack Evaluation Function** in **Section 3.2.1**.

1. Deepfool Variants(**Section 3.1.0**)
   (a) Pure Deepfool (**Section 3.1.1**)
   (b) Spiked Deepfool (**Section 3.1.2**)
   (c) QGA Deepfool (**Section 3.1.3**)
2. Single Attack Evaluation Function (**Section 3.2.1**)

4.3.1 Pure Deepfool

Following the approach of the original code and several other examples online, the original Deepfool algorithm was successfully implemented. This study utilised some additional benefits of object oriented programming to program the attack as a class rather than a simple function. This was decided in order to provide more functionality to the classes in future studies by extending the class and its methods.

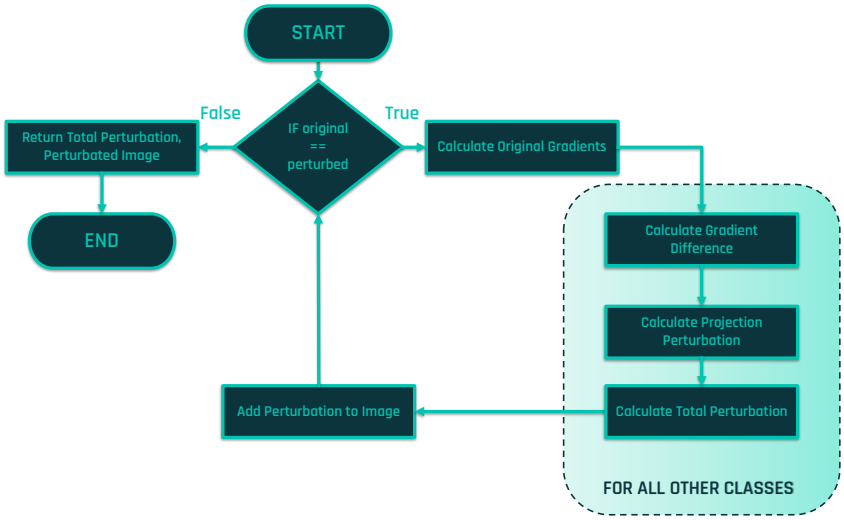

Figure 4.2: Flowchart representing basic operations of traditional Deepfool.

Figure 4.2 shows a flowchart of the basic operations that the pure Deepfool algorithm proposes. A similar approach was used to implemented Deepfool by modularising its design during development. Additionally, an important distinction must be made again in regards to how Deepfool works. As instructed by the original paper's repository[23], in order for Deepfool to work, the algorithm expects **logits** as output, hence we can either slice the model before activation or simply build it without **softmax** activation. In this study we used the latter option to train the models without a softmax activation, hence linearly activating the output to get the logits. In order to present results later on, softmax is simply used in a separate function called *decode_logit_preds(...)*, which requires the models output and outputs normalised probability outputs.

4.3.2 Spiked Deepfool

Using the traditional Deepfool as a basis and the additional random perturbation discussed in Chapter 3 and theorised in Figure 3.8, the newly introduced code for the spiked variant is added as a few additional lines at the end of the main loop of the Deepfool algorithm. This means that at each main iteration of the loop the algorithm determines if a random perturbation must be added to the total perturbation calculated.

[23] `https://github.com/LTS4/DeepFool/tree/master/Python`

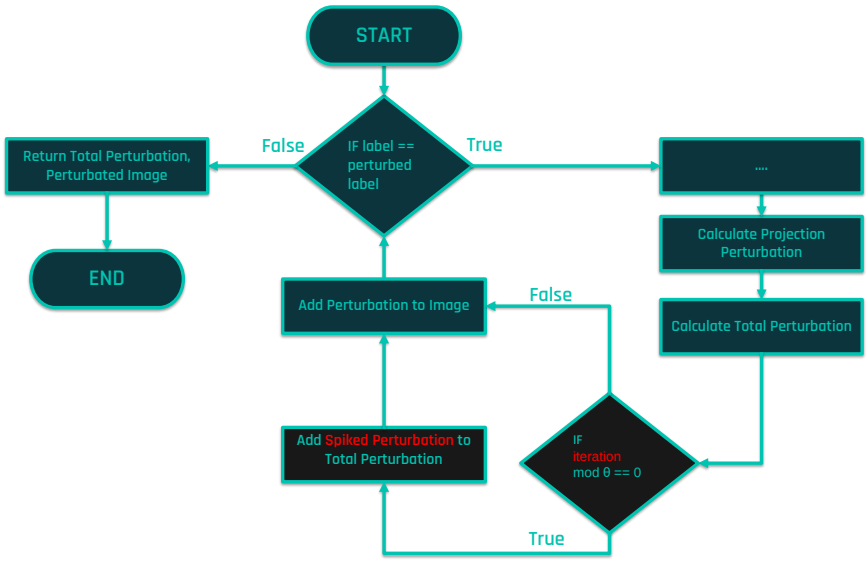

Figure 4.3: Flowchart representing the additional perturbation of Spiked Deepfool.

Figure 4.3 demonstrates how the additional random perturbation is added to the original algorithms end with the help of a flowchart. The main part of the algorithm is omitted in the process block noted by *(...)*, while the essential processes that come before and after the added code block are kept intact in the diagram. Attention should also be pointed on how the spiked perturbation must be added to the total perturbation before it is added to the image, since if that step does not take place then a prediction would be made for a different total perturbation at the end of the algorithm which would be wrong.

4.3.3 QGA Deepfool

The QGA Deepfool variant has been implemented similarly to the previous two attacks, based on the original code. To understand how to implement this variant, it is necessary to grasp the original code's important steps. Having said that, in order to calculated the proposed additional gradient points, this action must be performed when the **gradient** for the **projected class** is calculated. Initially, an iterative method of calculating additional gradient points was used, but was quickly dropped due to complexity increase concerns. Instead it was deemed much safer and faster to simply create an array of adjustments which can be applied to the first gradient calculated, thus creating five total gradient points. Then, we simply need to change the subsequent processes that calculate the perturbation needed to project the image to the specified class, to do that iteratively for all gradient points.

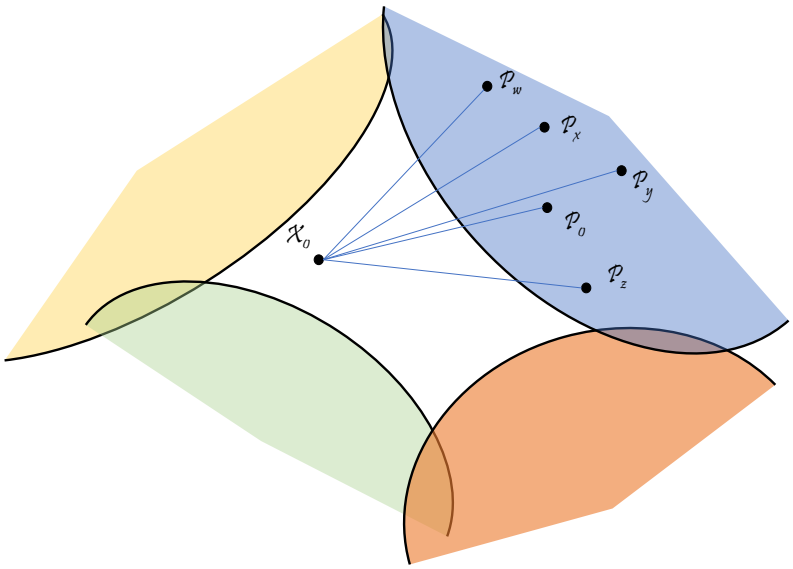

Figure 4.4: Linear approximation based on gradients in QGA Deepfool.

Figure 4.4 demonstrates how the envisioned variant functions in terms of gradient linear approximations. The diagram shows colour defined areas for some arbitrary class decision boundaries other than the original class and sample X_0 of which its gradients must be calculated. The points shown in the light blue decision boundary are examples of gradients calculated in a single inner iteration of Deepfool where P_0 is the current gradient calculated as practiced in traditional Deepfool. The other 4 points are products of additional gradient calculation where a is an adjustment and $a \in A = \{w,x,y,z\}$ for which $P_{(a)} = P_0 \times a$. The straight lines serve as a representation of the linear approximation between the two point's gradients.

4.3.4 Adversarial Robustness Testing

Last but not least, the final part of this study's implementation is the adversarial robustness test and evaluation code. In order to evaluate the adversarial robustness of each of the three models for the three variants, a function that performs iteratively the attacks on the whole dataset was devised. The function can also calculate the mean execution time for each attack on the test samples, while the mathematical representation of the adversarial robustness formula is introduced in the next chapter.

1. Adversarial Robustness Calculation Function (**Section 3.3.1**)
2. Pure Deepfool Robustness Evaluation (**Section 3.3.2**)
3. Spiked Deepfool Robustness Evaluation (**Section 3.3.3**)
4. QGA Deepfool Robustness Evaluation (**Section 3.3.4**)

Chapter 5

Evaluation

In this chapter we perform the most important part of this experiment, which is testing and evaluation of our model and adversarial attacks. A structured analysis of the various evaluation methods used will follow, in order to determine the degree of success of this experiment according to the set goals and questions. As part of the evaluation we will partake in certain strategies previously performed by the authors of the original Deepfool algorithm based on their research of relevant literature [39].

5.1 Testing Environment

Figure 5.1 lists the specifications of the machine that this project's development and evaluation took place on. We mostly list the relevant specifications that had a foreseeable impact on the performance of evaluations, such as operating system and version, the processing units, be it CPU or GPU and its subsequent speed and memory specifications as well as disk storage and physical memory.

TESTING ENVIRONMENT SPECIFICATIONS	
OS	Windows 10 Pro
OS Version	(20H2)
Architecture	64-bit
CPU Model	AMD Ryzen 5 1600
CPU cores	Six-Core Processor
L2 Cache Size	3072
L3 Cache Size	16384
CPU Speed	3.20 GHz
Disk	Western Digital Blue NVME
Disk size	1.0 TB
GPU	NVIDIA GeForce RTX 2070 SUPER
RAM	16.0 GB
RAM Frequency	3200 MHz

Table 5.1: Developing and Testing Environment Specifications

5.2 Model Testing and Evaluation Overview

To ensure that the later evaluation of adversarial attacks which will follow is successful we must first ensure that a proper evaluation of the models tested is performed. In order to achieve that, we are obliged to present certain evaluation and testing methods as well as general information that is essential for the reader to know. This not only can facilitate a better understanding of the experiment this project has attempted, but it will also aid future implementations and researchers follow in these footsteps if this study could spark further interest.

5.2.1 Evaluation methodology

Here, we will discuss the methodology followed in terms of evaluation in the order that it will be presented in this section shortly. We divide the evaluation into subsections for each of the

three models that were tested, namely Xception, Inception V3 and EfficientNet-B0. Each model's evaluation will be discussed in its corresponding metric section. We decided to evaluate each of the models with the accuracy and validation accuracy curves, confusion matrix as well as the classification report of other metrics which we will go into further details below.

We should also clarify that in the sections that follow we will be referring to the classification classes with numbers contrary to the exported confusion matrix and classification report that use the string format real name of each class's identity. To avoid confusion we provide a short list of the identities in the correct enumerated order below in Figure 5.2.

Classes	
Class 1	Amy Adams
Class 2	Antonio Banderas
Class 3	Ashley Olsen
Class 4	David Hasselhof
Class 5	Denzel Washington
Class 6	Russell Brand

Table 5.2: Class numbers corresponding to the 6 chosen identities.

5.3 Training and Validation

As already established above, this section is responsible for presenting the plotted curves of training and validation accuracy/loss metrics. By doing so we can infer useful information about the training process as well as determine whether changes could have been made prior to or during training to improve our methods. Such changes could refer to weight regularisation techniques, hyper-parameter tuning and learning rate decay strategies. Finally we will be reviewing each of the model's miss-classified images to determine whether bad implementation practices are among the factors that affected classification. We perform the above steps for each of tested models as well as present other types of evaluation which follow after this section, before we attempt to finally evaluate the adversarial attacks on them and calculate adversarial robustness.

5.3.1 Xception model Accuracy/Loss

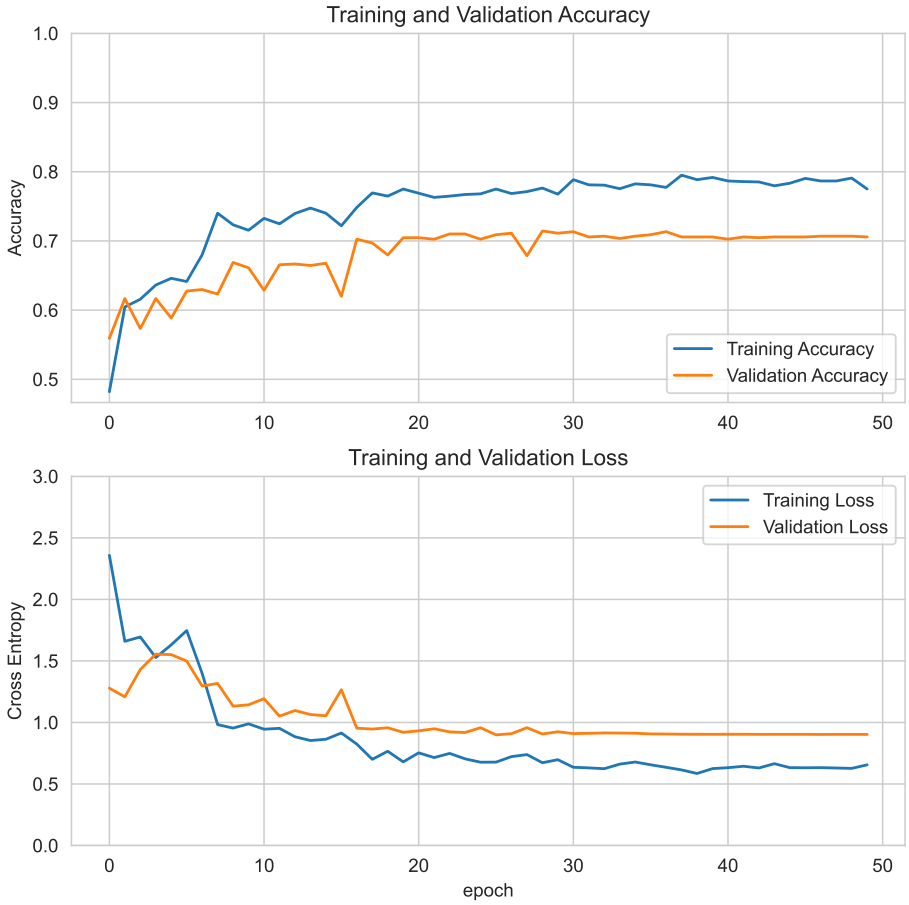

Figure 5.1: Training and validation loss/accuracy metrics plotted against increasing training epochs before fine-tuning (frozen CNN) for the Xception model.

Figure 5.1 shows the training and validation accuracy/loss for the Xception model before fine-tuning. We trained our model for 50 epochs, down from 100, given that no significant increase was noticed in prior test runs. Evidently, it is noticeable that before fine-tuning, hence with the main body of the model and its weights frozen, the validation accuracy plateaus at around 70% while the training accuracy continues to increase up to 99-100%. This is indicative of slight overfitting on the added layers on the top of the model. As suggested by most sources found online, that is not very alarming, given that a lot of improvement will be demonstrated after fine-tuning. Nevertheless, it does raise concerns of a possible glass ceiling to our models final validation accuracy.

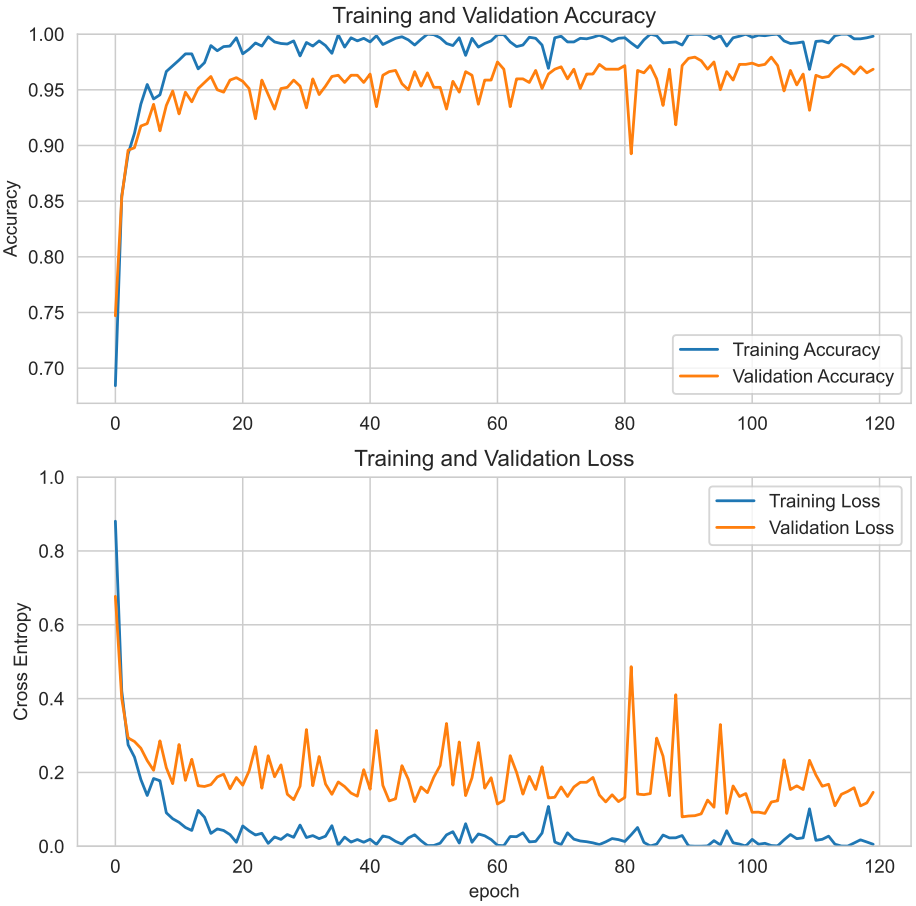

Figure 5.2: Fine-tuning training and validation loss/accuracy metrics plotted against increasing training epochs for the Xception model.

After fine-tuning the model for 150 epochs we notice a great increase in validation accuracy, while accuracy and loss remained stable at good percentages. It is clear that after unfreezing the layers, the overfitting becomes less with increasing training epochs. This can be attributed to the fact that the inner feature extraction capabilities of the once frozen layers and their weights is now being better adjusted to extract more meaningful features which are closer to our domain of face recognition, while also discarding feature extraction capabilities for other areas of ImageNet that are not needed for our classification problem. The model was trained until a satisfactory accuracy was observed which for this model was 97% validation accuracy. Nevertheless, it could be the case that with slightly more epochs, an even higher validation accuracy could potentially be attained, given that accuracy did show some very small increase steps, even though extremely small in nature. Regardless, we believe that 97% validation accuracy is more than satisfactory for what we plan to evaluate further ahead.

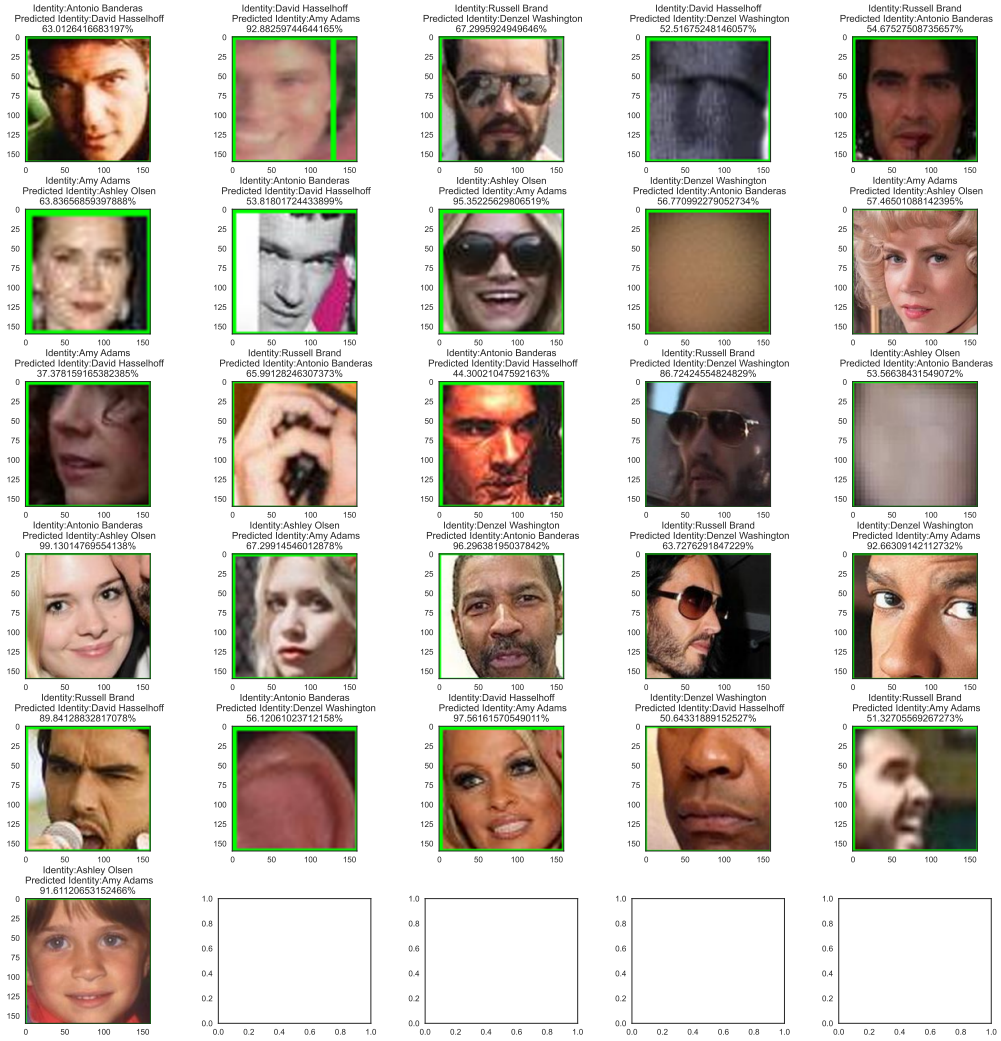

Figure 5.3: Miss-classified images in the test set for the Xception model.

Figure 5.3 shows all the miss-classified faces in the test dataset, plotted with labels of their original identity and the probability of their highest predicted identity. Out of 921 sample images only 26 were miss-classified. Upon closer inspection we can clearly see that some images where in fact incorrectly classified due to not properly handled by the face extraction process. To elaborate further, 8 images out of the total 26 miss-classified examples did not have their face properly extracted, while 4 other images out of the 21 left, show a face with sunglasses, which clearly affected the recognition process. From the remaining images, it is also observed that the some had more complicated poses, leaning sideways or looking down, which undoubtedly affected recognition. This is indeed possible given that such complications in pose could affect the extracted features inside our model, hence affecting the overall class probabilities generated. Another important downside noticed is present in the 3rd image of the semi-last row where the face extracted is not the actual face of the proposed class. This is undoubtedly a result of bad image samples, given that an image with multiple faces was used and resulted into this false face extraction.

5.3.2 InceptionV3 model Accuracy/Loss

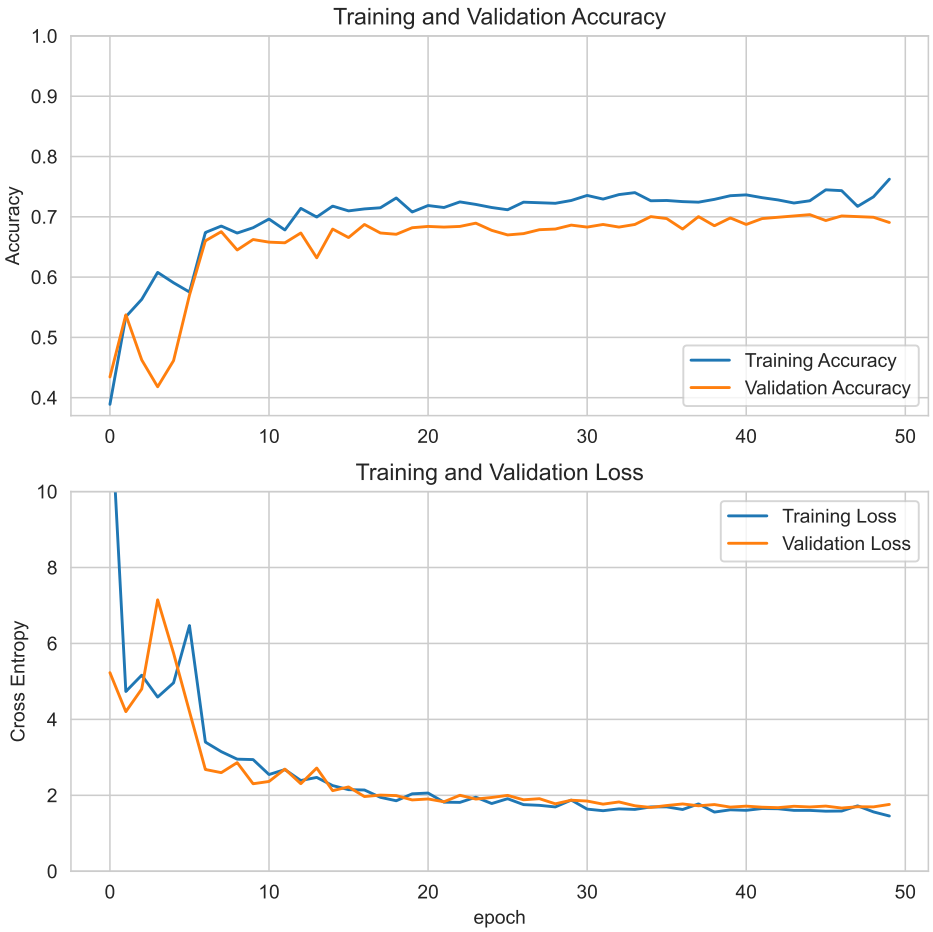

Figure 5.4: Training and validation loss/accuracy metrics plotted against increasing training epochs before fine-tuning (frozen CNN) for the Inception v3 model.

Figure 5.4 shows the training and validation accuracy/loss for the second model trained before fine-tuning, an Inception v3 model. The model was only trained for 50 epochs before fine-tuning, since the accuracy and loss metrics were observed to plateau at epochs higher than that. Validation accuracy before fine-tuning reached a maximum of 70% while loss was relatively high as well. This of course should not be alarming since both metrics will be normalised after fine-tuning when layers and weights are unfrozen.

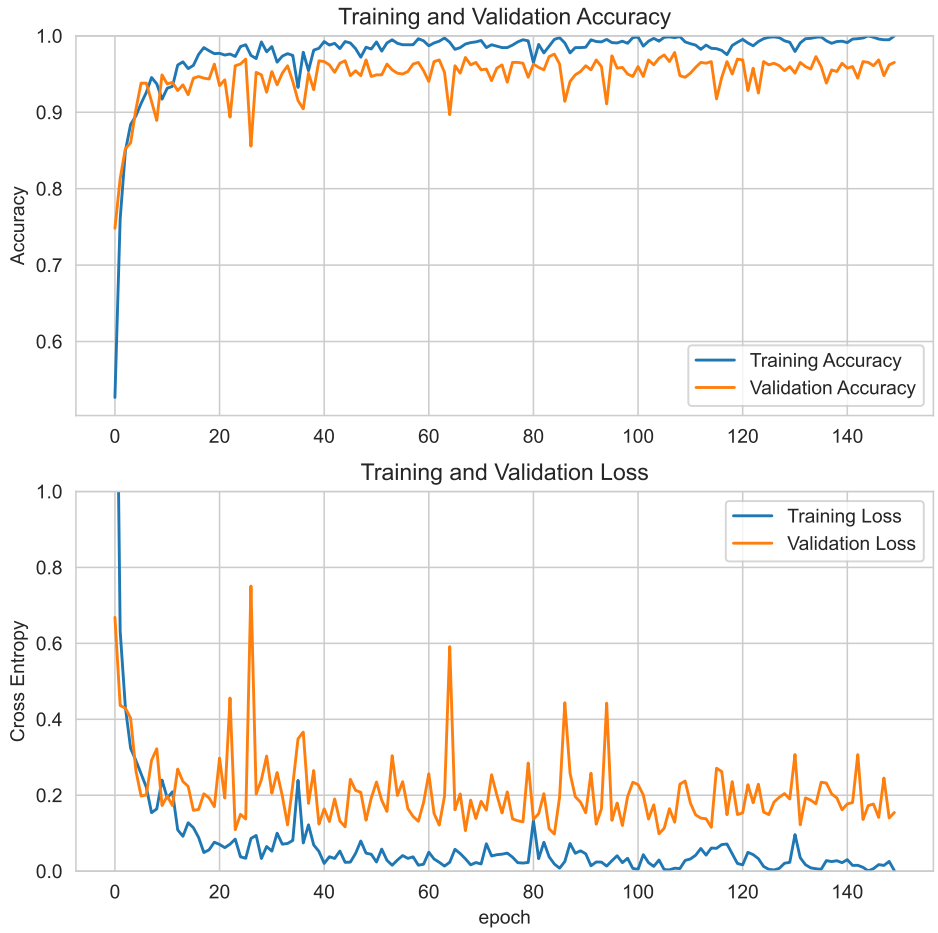

Figure 5.5: Fine-tuning training and validation loss/accuracy metrics plotted against increasing training epochs for Inception v3 model.

The slight overfitting noticed in the Xception model compared to Inception v3 before fine-tuning, is less noticeable in this model, yet the difference is still negligible. Figure 5.5 shows the training and validation accuracy and loss after fine-tuning the model. We fine-tuned the model for 150 epochs and noticed a great increase in both metrics at a very rapidly increasing rate. Once, again it is evident that after unfreezing the layers the overfitting problem is quickly minimised. Given that the Inception v3 model, has quite a different inner architecture, it seems that the model's feature extraction capabilities after unfreezing the layers and their weights are becoming even better but show a different learning pattern than Xception while gradually becoming attuned to the new classification problem. The models showed a surprisingly similar final validation accuracy with the Xception model at 97%. Be that as it may, we might have seen even greater model performance if we could apply some learning rate optimisation techniques, even though fine-tuning is a rather difficult process to optimise in transfer learning to converge to better results. Despite that, the model's final metrics in terms of validation accuracy and loss seem to be more than fine to perform the experiments planned ahead.

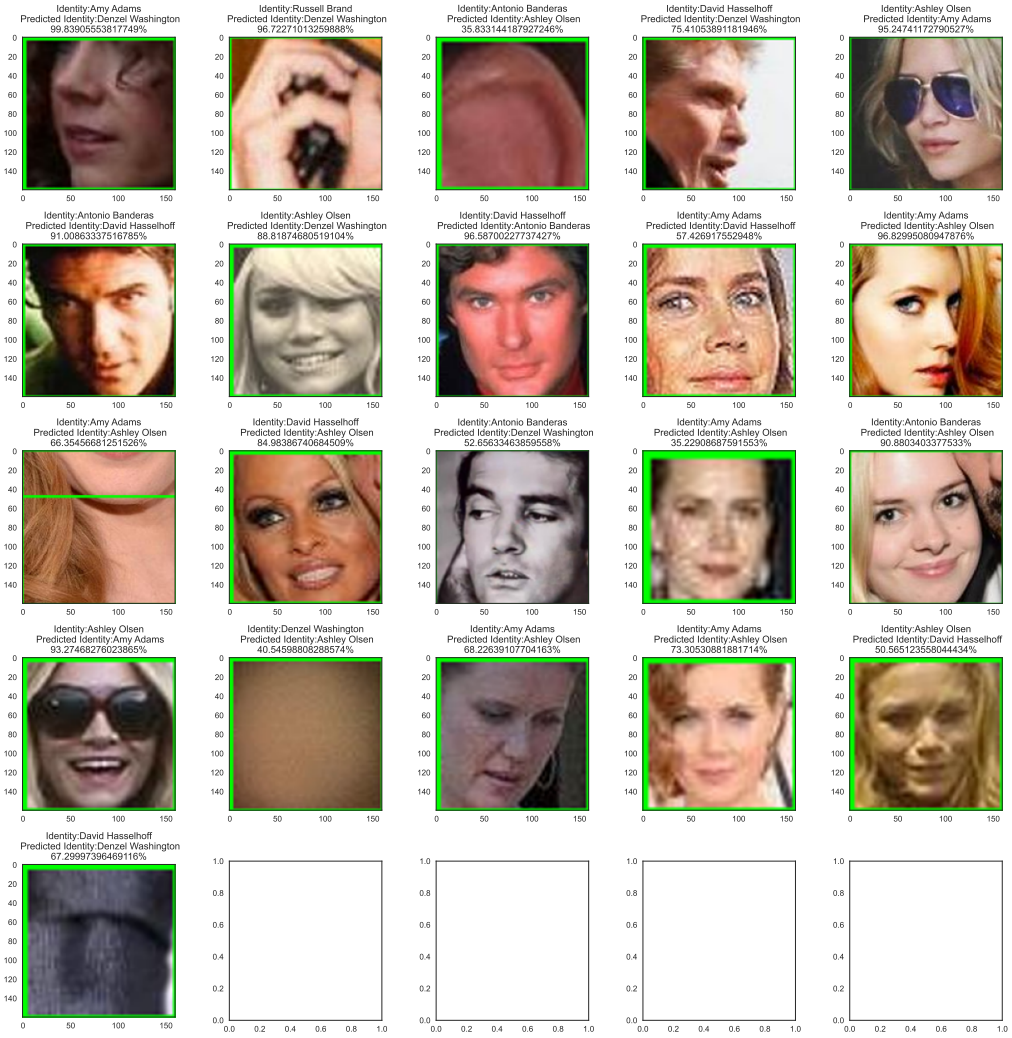

Figure 5.6: Miss-classified images in the test set for the Inception v3 model.

Figure 5.6 shows the plotted images of miss-classified samples of the test set. The Inception model with its slightly higher validation accuracy managed to miss-classify even less images of the test set compared to the Xception model, at 21 miss-classified samples. Looking at the miss-classified samples, we can see that 3 samples had either glasses or obstructed eyes, which definitely affected feature extraction of meaningful features on the eye level of the face. From the rest of the samples we observe that 6 images were not suitably prepossessed in terms of face extraction, since many of them had no face in the image square, while 2 image samples had the wrong face extracted due to the image having multiple faces. Finally, we notice that around 4 miss-classified images, had non-frontal poses which undoubtedly made recognition more difficult, yet managed to perform better than Xception in that regard. Looking at these miss-classified samples, the conclusion that the dataset should have been properly cleaned from outliers and reviewed after pre-processing to check for unrecognised faces, is reinforced.

5.3.3 EfficientNet-B0 model Accuracy/Loss

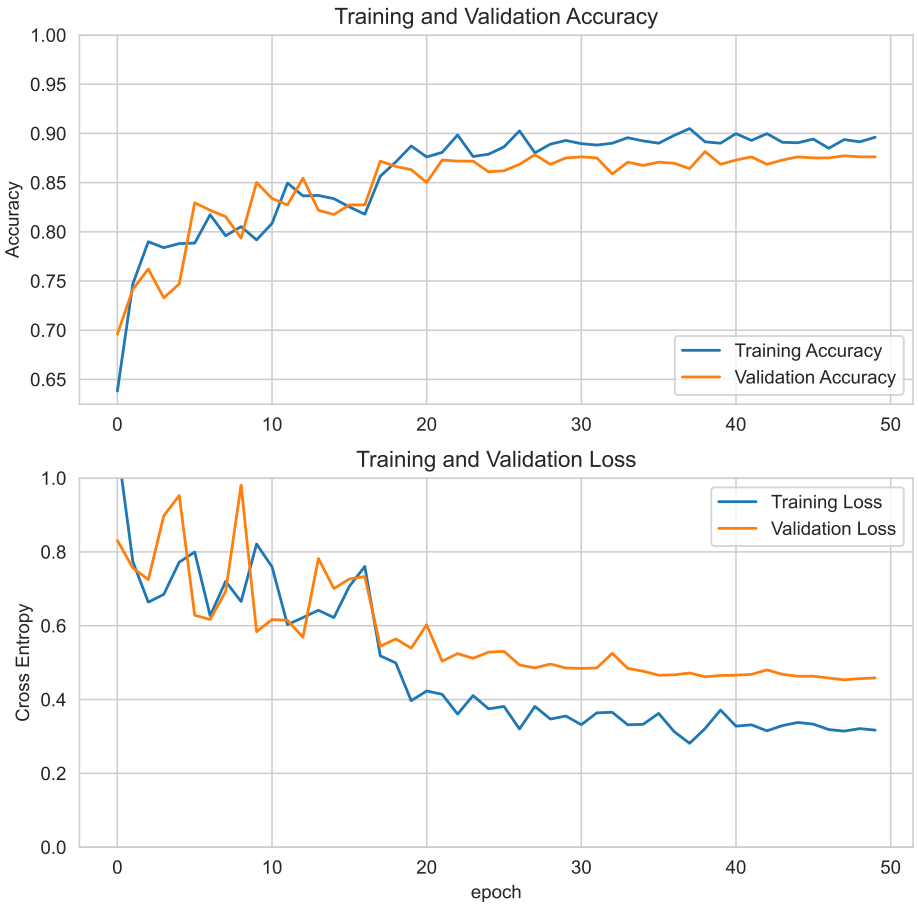

Figure 5.7: Training and validation loss/accuracy metrics plotted against increasing training epochs before fine-tuning (frozen CNN) for the EfficientNet-B0 model.

The last model to evaluate is EfficientNet-B0. In Figure 5.7 we can see the plotted accuracy/loss for training and validation before fine-tuning the model. It is evident that this model performs quite well from the start, since it has managed to converge much faster and at a higher validation accuracy (86%) than the previous models. Training was also considerably faster since EfficientNet is a relatively new and improved architecture with noticeably fewer internal parameters.

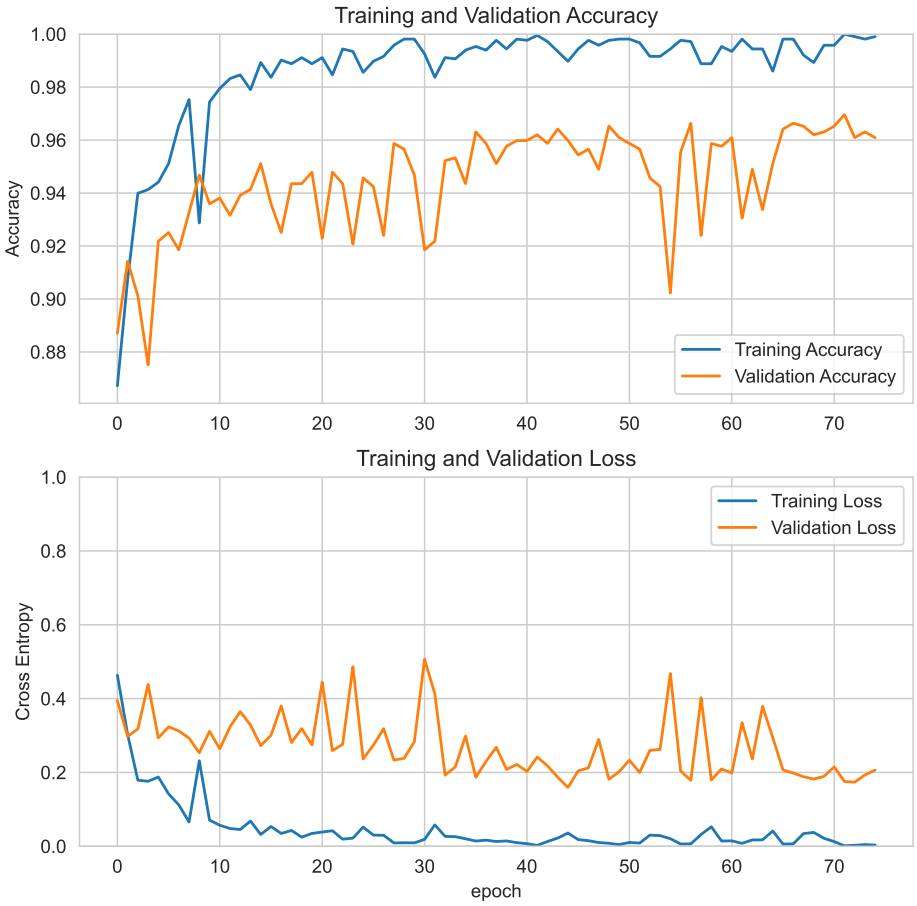

Figure 5.8: Fine-tuning training and validation loss/accuracy metrics plotted against increasing training epochs for the EfficientNet-B0 model.

After fine-tuning the EfficientNet model, we can clearly see a very similar performance with the two previous models in terms of training and validation accuracy/loss. What is different from the other two models however is that the model converged faster and thus less epochs were needed for the model to reach the same metric performance of the other models. This can be seen in the graph in Figure 5.8, since the graph shows the performance of the model while training for just around 80 epochs. The model has reached around 96-97% validation accuracy, which is roughly the same performance as the other models, at least in regards to training and validation accuracy and loss. The rest of the metrics and performance of the model will be showcased in their respective sections.

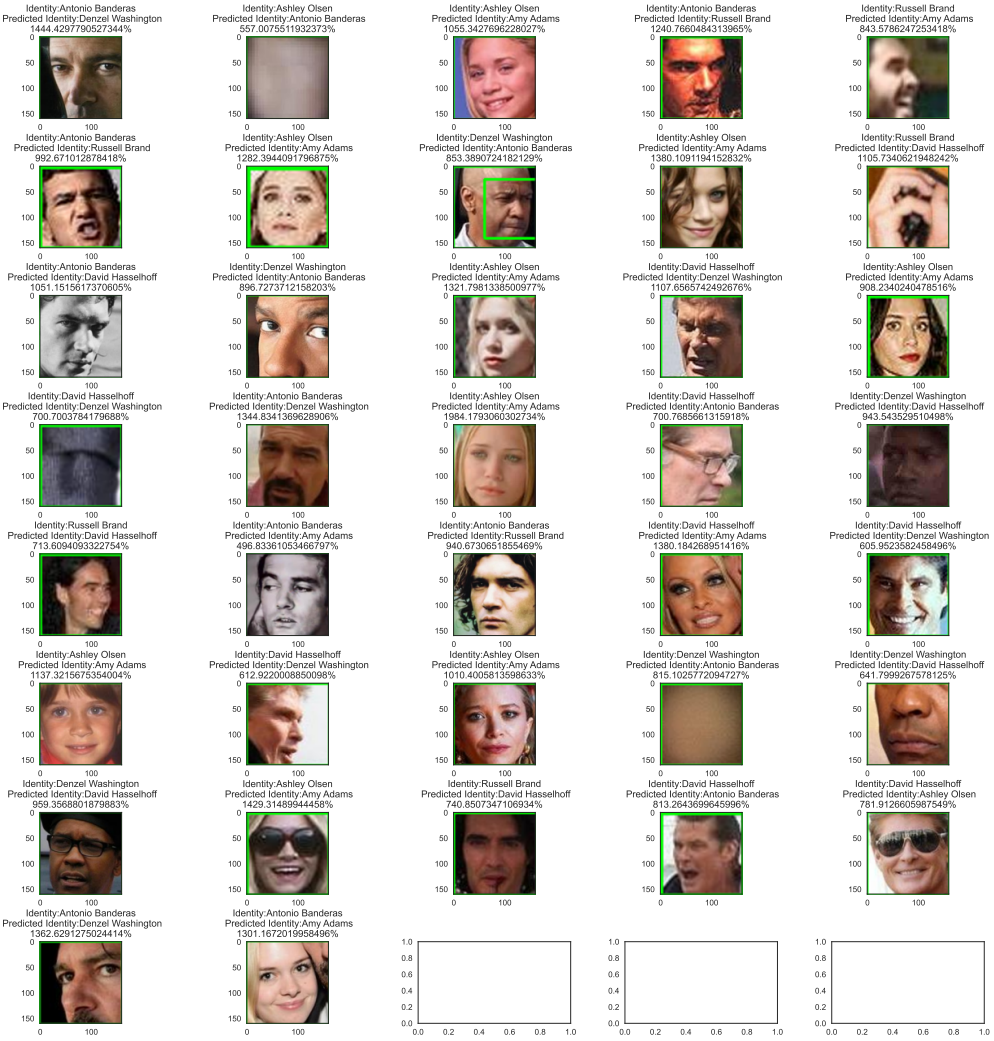

Figure 5.9: Miss-classified images in the test set for the EfficientNet-B0 model.

Figure 5.9 shows the miss-classified samples of the test set in the EfficientNet-B0 model. Quite surprisingly, even though the model seems to perform similarly to the other models, more images were miss-classified with in EfficientNet, given that we have plotted 37 samples. Once again, we can see that reasons such as pose variance, age and improper face extraction are primarily the cause of this miss-classifications. Out of all the samples reported, 4 images had no visible face in the frame, which points to the lack of examination on preprocessed images. The majority of the miss-classified samples are images which have somewhat unorthodox poses, with the head of the person in the picture not looking directly in front by having a very tilted head pose. Last but not least, 1 image could be miss-classified due to being a very young image of the identity class, while another image was mistakenly used in the place of the correct face as a result of multiple faces in a single image. It is quite interesting that unlike the other two models, EfficientNet seems to be better at classifying images with obstructed eyes, given that only 3 images included a face wearing glasses.

5.4 Confusion matrix

As part of our model evaluation, we also employed additional evaluation methods such as a confusion matrix for each of the models, provided by the scikit-learn library for machine learning [46]. A confusion matrix is a 2-D table that demonstrates the performance of our model on a given sample set (preferably not used in training). The performance is evaluated by calculating how the model prediction performs on the sample test by comparing the true labels with the annotated labels and then populating the matrix with the results for each of the classification problem's classes. The matrix is also colour coded, with a colour range explaining the importance of each colour. An adequately trained model should produce a matrix with a visible diagonal line, which demonstrated that most predictions for each class where in fact accurate. Hence, we will look at the generated confusion matrix for each of the models and discuss the observable results. Since we are already showing miss-classified examples in the previous section it is only natural to look at the confusion matrix next to understand how these miss-classifications are distributed among classes to determine whether there might be a sample bias between classes.

5.4.1 Xception confusion matrix

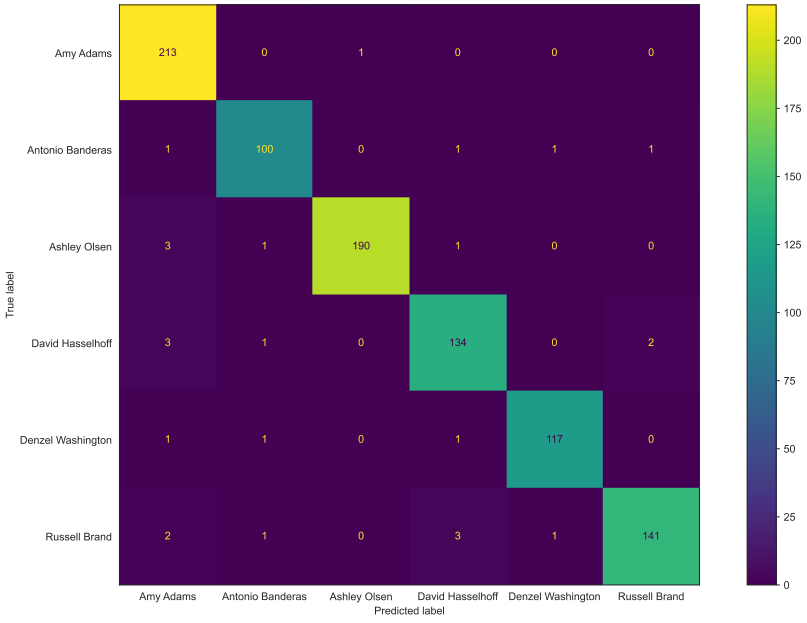

Figure 5.10: Confusion Matrix for the Xception model.

Continuing our evaluation of the Xception model from the previous section, we now move on to discuss the confusion matrix generated. A visible diagonal line across the matrix is shown, which definitely shows good results for our model. On the other hand, the line seems to be unevenly coloured, which indicated that even though we have good prediction performance for the model, the test set is not evenly distributed among the classification classes, given that the first class has 213 correctly predicted images whereas the second has 100. Nevertheless, it is interesting that the

1st class which also has the highest number of samples in the test set, has also the lowest number of miss-classified images at 1. This can also be attributed to the fact that the class in question has 689 total images, the 2nd highest number of samples of all classes, and since 214 were in the test set, means that a significantly higher number of good samples were also used during training, which might explain the overall good prediction rates for this class.

On the other hand, if we are to look at the highest number of miss-classified predictions, the 4th class comes first by a large margin, since it fails to predict correctly for 10 samples. This comes as no surprise as we have already seen many of these miss-classified examples in the previous section were many of them wore glasses, had uneven poses or wrongly extracted faces. Thus, this means that the 4th class and its predictions is hindered by a large number of bad image samples and highlights the necessity to clean the dataset from outliers before we even load it. Overall the Xception model seems to be performing very well given the nature of this confusion matrix.

5.4.2 InceptionV3 confusion matrix

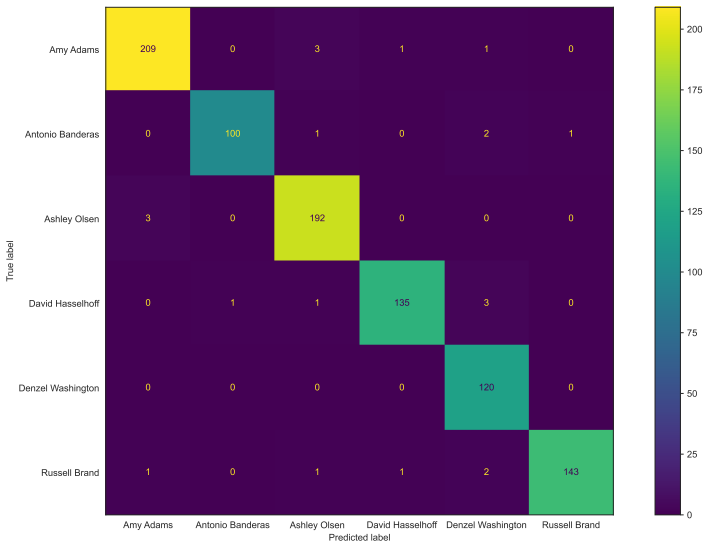

Figure 5.11: Confusion Matrix for the Inception v3 model.

Moving on to our Inception v3 model, we will now present the results shown from the confusion matrix in close relation to the results presented in the previous section where we discussed training, validation and miss-classified examples. Just like with the Xception model, we can clearly see a visible diagonal line across the matrix, with the same coloration, which bolsters the fact that this model has indeed been trained on a satisfactory level. On the other hand, we see that there are slight differences in the values of the confusion matrix as well as a different distribution between miss-classified examples, even if some remained the same, as established by the overview of miss-classified samples.

We can clearly see that classes 1, 4 and 6 had equally high miss-classified samples, while it is interesting that the 1st class had so many, given that it has the highest number of samples in the test dataset. Once again, it is not surprising that the 3rd class had great predictions, since

it had the highest number of samples across both datasets together and most importantly during training. The Inception v3 model seems to be great at predicting classes 4 and 5, while it predicted correctly all image samples for class 5. Finally, the uneven coloration of the diagonal line in contrast to miss-classified sample colours, shows that even though we have good predictions, the test dataset should have been more evenly distributed among the classes.

Furthermore, the confusion matrix confirms that the dataset contains a lot of bad image samples and requires cleaning to eliminate these miss-classifications. Taking everything into consideration, according to the confusion matrix, the Inception v3 model shows great performance and should be more than suitable to test the adversarial attacks against it.

5.4.3 EfficientNet-B0 confusion matrix

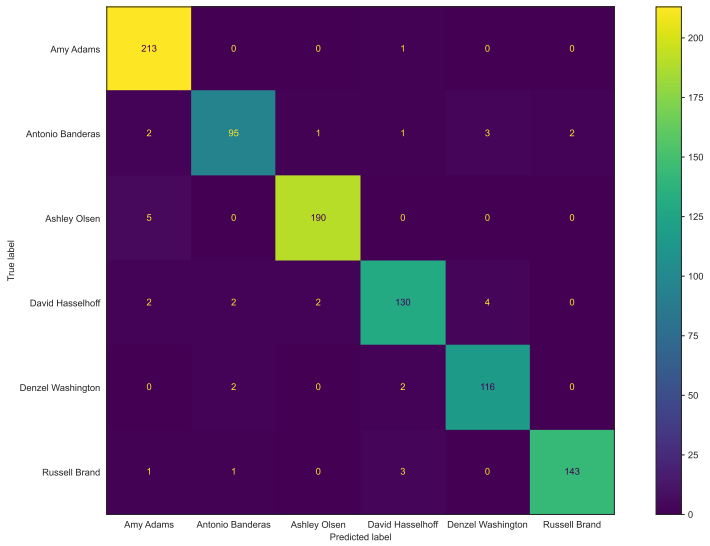

Figure 5.12: Confusion Matrix for the EfficientNet-B0 model.

Moving on the confusion matrix of the EfficientNet-B0 model in Figure 5.12, we clearly see yet again that a visible diagonal line is shown on the matrix, despite the uneven coloration. Evidently, class 1 seems to have the hightest number of correctly predicted samples, followed by class 3. In terms of miss-classifications, a higher number of wrong predictions is observed in class 2 and 4, while class 1 has the also the least amount of wrong predictions. All in all this confusion matrix demonstrated better the distribution of bad predictions noticed in the plotted miss-classified samples graph in Figure 5.9. we must however look at other metrics as well in order to extract more meaningful conclusions for the performance of this model.

5.5 Classification report & additional metrics

A classification report is a method of evaluating a model by the scikit-learn library [46]. It calculates and generates a textual report for a number of different metrics which can be used to assess a models performance, other than just training accuracy. This is an essential test to perform as some results can only be observed by looking at these values. Thus, we will perform this task

of generating a classification report for each of the tested models, to evaluate their performance. Below you will find a list of descriptions of the different metrics used in each report, since it is important to introduce each of these metrics to better understand their underlying value in this model assessment. The formulas for each of these metrics however, will be attached in the **Appendix A** to keep a balanced size for this chapter.

Precision - The classifiers ability to correctly avoid yielding a positive when it should yield negative, meaning the ratio of true positives over the total of true and false positives. Hence, it is the percentage accuracy of correct predictions.

Recall - Recall stands for the degree that positive samples are predicted as true positives. We do that by using the ratio of true positives over the total of true positives and false negatives.

F1 Score - The F1-score is a measure used to assess the quality of our classifier. It is defined as the harmonic mean of the precision and recall.

Support - Support stands for the number of samples of the true class that lie in the sample set/dataset, or simply the total number of samples per class.

Accuracy - Accuracy denotes the ratio of correct predictions over the sum of the total images and is also commonly referred as classification accuracy, a common metric used when training a model.

Macro Average - Macro Average stands for the overall performance of the classifier on the dataset for each of the other metrics used, like precision, recall and f1-score.

Weighted Average - Similarly, Weighted Average calculates the average score by also introducing weights. To calculate weighted average, each sample in the data set is multiplied by a predetermined weight before the final calculation is made. This in turn will might provide more insight than macro average in specific cases.

5.5.1 Xception metrics

	precision	recall	f1-score	support
Amy Adams	0.96	1.00	0.97	214
Antonio Banderas	0.96	0.96	0.96	104
Ashley Olsen	0.99	0.97	0.98	195
David Hasselhoff	0.96	0.96	0.96	140
Denzel Washington	0.98	0.97	0.98	120
Russell Brand	0.98	0.95	0.97	148
accuracy			0.97	921
macro avg	0.97	0.97	0.97	921
weighted avg	0.97	0.97	0.97	921

Figure 5.13: Classification report of different metrics for the Xception model.

As you can see in Figure 5.13, the exported classification report for the Xception model is presented. The figure lists the observable points of interest in regards to specific class metrics. The lowest precision is observed to be in the 1st, 5th and 4th class (96%), while the highest in the 3rd (98%). In regards to recall we see that the lowest is for class 6 (95%) while the highest for class 1 (100%), which is astoundingly good. Once again, this is not surprising given that class 4 had a high amount of bad predictions due to bad samples (pose, glasses, no face), whereas class 1 had a very high number of samples throughout the training and test dataset with a fair amount of sample quality. If we look at the F-1 score we can see that class 4 is equally low as class 2 (96%), given that they have the same harmonic mean of precision and recall. The highest F1-score is that of class 3 and 5. Overall the model seems to be performing rather well given that the accuracy, weighted and macro average are at 97%.

5.5.2 Inception metrics

	precision	recall	f1-score	support
Amy Adams	0.98	0.98	0.98	214
Antonio Banderas	0.99	0.96	0.98	104
Ashley Olsen	0.97	0.98	0.98	195
David Hasselhoff	0.99	0.96	0.97	140
Denzel Washington	0.94	1.00	0.97	120
Russell Brand	0.99	0.97	0.98	148
accuracy			0.98	921
macro avg	0.98	0.98	0.98	921
weighted avg	0.98	0.98	0.98	921

Figure 5.14: Classification report of different metrics for the Inception v3 model.

Figure 5.14 shows the classification report for the Inception v3 model. Compared to the Xception model we see on average similar results, yet if we look at the different metrics individually we can deduce how each model performs differently on each class of the dataset. This can be demonstrated by looking at the precision of this model as contrary to the previous model examined, the lowest score is that of class 5 (94%), while the highest is that of classes 2, 4 and 6 (99%). In terms of recall, the lowest was observed in classes 2 and 4 (96%) and the highest in class 5 (100%). The F1-score helps us examine insights on precision and recall altogether and doing so we see that the lowest F1-score is that of classes 4 and 5 (97%). This is in fact interesting since class 5 had the lowest precision but also the highest recall (94 vs 100%). The macro and weighted average for the Inception model share the similar results to the other models since we are using the same dataset. An average of 98% F1-score and accuracy is measured, which is slightly better than the Xception model. As a result, it is believed that the Inception model has also been sufficiently and properly trained for us to contact further experiments about adversarial robustness.

5.5.3 EfficientNet-B0 metrics

	precision	recall	f1-score	support
Amy Adams	0.96	1.00	0.97	214
Antonio Banderas	0.95	0.91	0.93	104
Ashley Olsen	0.98	0.97	0.98	195
David Hasselhoff	0.95	0.93	0.94	140
Denzel Washington	0.94	0.97	0.95	120
Russell Brand	0.99	0.97	0.98	148
accuracy			0.96	921
macro avg	0.96	0.96	0.96	921
weighted avg	0.96	0.96	0.96	921

Figure 5.15: Classification report of different metrics for the EfficientNet-B0 model.

Finally, the classification report with other metrics exported for the EfficientNet-B0 model is shown in Figure 5.15. We observe the highest precision in class 6 (99%) and the lowest in class 5 (94%). The highest recall was observed in class 1, which achieved perfect score (100%), while the lowest recall was measure in class 2 (91%). It is also worth noting that the majority of classes performed well in terms of recall. Looking at the metrics from a wider perspective using F1-score, we see that the highest achieved score was recorded by class 6 (98%) whereas the lowest score in classes 2 and 5 with (94%). All in all, if we look at the macro and weighted average, this model has also demonstrated a sufficiently good performance at 96%, making it viable for evaluating the adversarial attacks that will follow.

5.6 Adversarial robustness evaluation

As mentioned in the literature overview, Moosavi-Dezfooli et al., proposed their own novel approach in calculating adversarial robustness of a model and used it in their evaluations [39]. The equation in Figure 5.16 shows their proposed approach. Let us go over the equation to eliminate any misunderstanding before we perform this type of evaluation on our Deepfool variants. The **average adversarial robustness** of the model noted by $\hat{\rho}_{adv}(f)$, is said to be represented by the sum of the norm of **minimum perturbations** $\hat{r}(x)$, for all samples over the **test set** Δ over the norm of **sample image** x of the test set, given that $x \in \Delta$, and finally divided by the number of samples in the test set.

Even though the original paper also included two more methods of calculating adversarial robustness from previous studies, they also conclude that their proposed approach is more accurate in detecting directions that could fool neural networks, due to the calculation of smaller perturbation vectors [39]. With this in mind and the original paper as evidence, we believe that the Deepfool approach of calculating adversarial robustness of a model alone, will serve as a fine evaluation method to yield results on both the quality of the model and the attack variants. Thus, in the subsections that follow we will test the adversarial robustness of each of the models with each of the Deepfool attack variants and report the model's test error, adversarial robustness and average time per attack as the original Deepfool paper attempted.

$$\hat{\rho}_{adv}(f) = \frac{1}{|\Delta|} \sum_{x \in \Delta} \frac{\|\hat{r}(x)\|^2}{\|x\|^2}$$

Figure 5.16: How to calculate adversarial robustness of a classifier by Moosavi-Dezfooli et al. [39].

After trying many possible configurations for the models to produce good results, some of them like the final Inception v3 model were really resilient to attacks, to the point that most images were intensely perturbed to miss-classify them. For instance, evaluating the QGA Deepfool adversarial robustness on Inception v3 took about 50 straight hours. Still adversarial robustness was calculated for all models albeit on lower iterations on Inception v3 since calculations were taking a lot of time to run.

5.6.1 Deepfool Evaluation

In Figure 5.3, we see the results of the Pure Deepfool attack on the three models. Evidently there are a lot of observations to unpack. We see that in terms of speed execution on traditional Deepfool, Xception seems to be launching attacks way faster than the other two models. It is followed by EfficientNet and lastly Inception v3, which takes on average approximately 3 times the time of EfficientNet B0 while almost 6 times that of Xception. In terms of average adversarial robustness, we see that Inception v3 shows a significantly higher adversarial robustness value. The adversarial robustness value for Xception also appears to be 3 times more than that of EfficientNet B0. These observations of the higher average time required, the higher adversarial robustness value as well as the slightly better model performance, might indicated that Inception V3 is more robust than the other two models.

Pure Deepfool adversarial robustness test			
Classifier	**Test Error**	$\hat{\rho}_{adv}$ [39]	**time (s)**
Xception	2.8%	9.036×10^{-3} (4 s.f.)	1.8352 (5 s.f.)
Inception V3	2.2%	3.079×10^{-2} (4 s.f.)	12.549 (5 s.f.)
EfficientNet-B0	4.0%	3.135×10^{-3} (4 s.f.)	4.3414 (5 s.f.)

Table 5.3: Pure Deepfool adversarial robustness test for each of the models.

5.6.2 Spiked Deepfool Evaluation

The results of the Spiked Deepfool, meaning the Deepfool variant with added random perturbations, seems to be performing faster the adversarial perturbations than the pure Deepfool attack. Despite, being a very naive addition to the original algorithm, the mean times for each adversarial attack are evidently highly reduced. Figure 5.4 shows these results of the Spiked Deepfool attack

on the three models. In terms of execution speed, Xception continues to demonstrate fast average times, followed by EfficientNet and Inception V3 at the end, with more than 4 times the average time of Xception. In terms of adversarial robustness on the Spiked Variant, EfficientNet performs the worse, followed by Xception while Inception appears to have the highest at almost twice the value of Xception.

Spiked Deepfool adversarial robustness test			
Classifier	**Test Error**	$\hat{\rho}_{adv}$ [39]	**time (s)**
Xception	2.8%	4.498×10^{-2} (4 s.f.)	0.7761 (5 s.f.)
Inception V3	2.2%	8.876×10^{-2} (4 s.f.)	4.3343 (5 s.f.)
EfficientNet-B0	4.0%	1.089×10^{-2} (4 s.f.)	1.5940 (5 s.f.)

Table 5.4: Spiked Deepfool adversarial robustness test for each of the models.

5.6.3 QGA Deepfool Evaluation

Figure 5.5 shows the results of measuring adversarial robustness of the three models for the QGA Deepfool variant. As mentioned already, the general and most obvious observation is the increase in average execution time for all models of this attack variant. It has been estimated that Inception takes more than twice as much time to execute an attack on average, compared to EfficientNet. Xception on the other hand, executes attacks even faster that EfficientNet. Similar to the traditional Deepfool and Spiked Deepfool attacks, the adversarial robustness of each of these models is quite similar throughout. Once again Inception v3 shows the highest adversarial robustness followed by Xception and last comes EfficientNet B0, at approximately 4 times less than Xception.

QGA Deepfool adversarial robustness test			
Classifier	**Test Error**	$\hat{\rho}_{adv}$ [39]	**time (s)**
Xception	2.8%	8.551×10^{-3} (4 s.f.)	4.9731 (5 s.f.)
Inception V3	2.2%	2.977×10^{-2} (4 s.f.)	15.726 (5 s.f.)
EfficientNet-B0	4.0%	2.600×10^{-3} (4 s.f.)	6.4363 (5 s.f.)

Table 5.5: QGA Deepfool adversarial robustness test for each of the models.

5.7 Requirement Evaluation

In this section we will perform a summative subjective evaluation of the research questions proposed at the beginning of this research. We re-introduce each of these questions and discuss how we addressed them as well as the degree of which they have been answered by this research.

RQ-1: *Can a fairly new architecture and collection of pipeline components be used to create a facial recognition model for evaluating adversarial attacks?*

We have managed to implement **three** models from **recent state-of-the-art** architectures, namely Xception, Inception v3 and EfficientNet-B0. We streamlined the process of building the facial recognition models using defined pipeline components linearly executable in a python Notebook with the help of a python interactive kernel. Judging from our evaluation of the three models prior to this section, we can safely say that all three models perform exceptionally well at around 96-97% on average on metrics such as validation accuracy, precision, recall and F-1 score, despite the presence of some bad samples in the dataset. Having said that, it is believed that **RQ-1** has been achieved as the models were deemed appropriate to continue the evaluation of attacks on them.

RQ-2: *Does adding small random perturbations in the **Deepfool** algorithm produce, on average, more minimal perturbations than pure Deepfool?*

The Spiked Deepfool attack variant has been created to address this question. It has been designed in such a way that it produces a random perturbation if certain calculated criteria are met. The key point is that the random perturbation takes place at the end of each iteration of the main Deepfool loop, meaning after each iteration calculates hyperplanes and weight differences for all classes except the current one. This allows the new variant to keep random perturbations from being excessively triggered, but also efficient enough so help find a minimum perturbation faster. The evaluation of adversarial robustness has shown no clear results in terms of robustness but it has shown a significantly less time required to find a minimal perturbed image. In the Inception V3 model, the Spiked Deepfool variant showed approximately a third of the attack duration of traditional Deepfool. A different test should be set up to investigate whether the Spiked Deepfool variant, although faster, produces a larger perturbation than necessary to generate an adversarial image.

RQ-3: *Can **Deepfool's** gradient calculations be altered/improved, so that it considers more gradient points at each iteration?*

To address this question a more sophisticated variant was proposed in the form of QGA Deepfool, featuring a technique named quadruple gradient acceleration. This variant is different than traditional Deepfool in the sense that it calculates four additional gradient points and to project the image in a decision boundary. This in theory allows the attack to choose the linear approximation which produces less perturbation. Evidence gathered during our evaluation show several insights regarding this variant. The adversarial robustness of the models against QGA Deepfool seems vastly smaller than the other two attacks. On the other hand, in terms of execution speed, this variant seems to be performing very poorly

compared to the other two attacks. At times, calculating adversarial robustness for this variant took many hours of continuous code execution, given that we have a large test set of 921 images. Further examinations would be needed to examine a more computationally optimised method of such approach using multiple gradient points and comparisons. Despite that, the attack variant does work as intended, albeit with poor performance.

Chapter 6

Conclusion

Last but not least, we discuss various topics regarding the overall evaluation and fruitful conclusions arising from it. Mentions of future work are included as well as possible areas where we should discuss improvement in all areas of this research.

6.1 Limitations

Even though we took many precautions and designed the research process to minimise any limitations arising from our study, it was impossible to eliminate all of them. In this section we discuss what limitations were observed in this research and what could have been done to avoid them.

6.1.1 Experimental Limitations

After demonstrating many of the flaws of our experimental methods in the evaluation, we should list these flaws and report how we could alleviate these limitations. One of the most striking limitations found, was the quality of our dataset samples. Although, the VGGFace2 dataset offers an immense amount of samples, the images used should have been cherry-picked to avoid any images with multiple faces, or inappropriate material that will not be processed correctly [12]. It was also a mistake to include extrapolated bad samples such as images with many bad attributes such as a combination of bad poses, angles and objects obstructing the face like sunglasses and hands. This is evident from our evaluation of miss-classified samples, since it could safely be said that many of them fall into categories of bad samples we mentioned earlier.

Having a clean import of data is undoubtedly important, yet it is also crucial to examine the image data after pre-processing as well. In our case we did not perform this procedure at all and it is clear to see that had it been implemented, we would have a much better model performance. This is quite obvious given that some images after pre-processing, had no face detected and that affected prediction and the overall model accuracy without a doubt. It should be noted that for this project's preprocessing, we used a common technique by OpenCV for face detection, despite the fact that other alternatives were present such as FaceNet's face detection, which sometimes shows more accuracy but is more computationally intensive [49, 65]. The reason behind this choice was primarily speed, given that FaceNet required the use and prediction capabilities of a Keras model for every face to be extracted from the whole dataset.

Overall, in regards to experimental limitations, this study has shown the effects of improper pre-processing for a large and diverse dataset with different poses, ages and image dimensions. Evidently, these limitations caused a significant amount of our images to be falsely predicted and should be addressed with greater care in future projects.

6.1.2 Implementational Limitations

As far as implementational limitations go, a few remarks can be made for this study. All models tested had experienced a slight amount of overfitting around 3-4%. Looking at what could cause this small overfitting, we see that even though we had a small dataset and at times we trained the model for too long, it is believed that the most probable reason for it was the presence of bad samples in both the train and test datasets. Furthermore, it is unlikely that it was caused by wrong pre-processing, given that we use packaged pre-processing functions as well as data-augmentation techniques.

Sources usually recommend and use dropout as a regularisation technique to reduce overfitting. The early hyper-parameter tuning experiments showed no difference with different amounts of dropout and so it was kept constant at 0.2 for all models. Given the Xception original paper's insights it might have been a good idea to increase the dropout regularisation after all, since training data was not particularly large. It would have likely taken more time to converge, but could result in even less overfitting and more steady training during epochs [17].

Last but not least, other than implementational limitations regarding overfitting reduction, it would seem that more thought should have been put into designing algorithmically better attack variants. To elaborate further, the complexity of the Deepfool attack variants massively hindered our evaluation due to the immense time required to produce adversarial robustness tests. One such example is the adversarial robustness calculation of the Inception v3 model on the QGA Deepfool attack, since it took hours to finish execution. This comes as no surprise given that the QGA Deepfool variant calculates not one but five total gradient points at each iteration of the inner loop. This means that if no minimal perturbation is found quickly in early iterations of the outer loop, the algorithm becomes quite resource intensive, due to its complexity. This is further proved by the execution times recorded for Inception v3 on all attacks.

6.2 Future work and Improvements

It is imperative for us to establish what could further be investigated in the future, to maintain a scientific and academic interest in this specific field of research. Thus, our improvements and future work revolve around the different limitations that we have come across in this journey.

Therefore, having talked about the limitations of this research in regards to the data, it would be highly beneficial to clean the dataset from bad image samples in regards to pose, age, image quality and face obstruction in future iterations of this research. Furthermore in the future, it would be a good precaution to inspect the image data after it has been pre-processed to ensure that preprocessing is indeed correct and no bad samples were included. Moving on from future improvements based on limitations, it would also be a great and desirable scientific story to explore, to evaluate the robustness of the model after it has been fine-tuned with adversarial examples. By doing this, we would actually complete the methodology outline followed by Moosavi-Dezfooli et al. in Deepfool's original paper, which we kept short due to this project's alternative focus on Deepfool variants, face recognition and modern architectures [39].

In terms of the models used, it would be very interesting to explore even better model architectures such as other versions of EfficientNet, as well as additional regularisation techniques like l1 and l2 kernel regularisation, which might have made adversarial attacks even more difficult on

the model. Last but not least, it would also be an interesting approach to further experiment and tweak Deepfool variants, to be able to compare with less effort with more recent and advanced adversarial attacks on image recognition classifiers.

6.3 Conclusion

To summarise everything covered in this research, we looked at the history and evolution of facial recognition from manual facial feature mapping to massive deep learning breakthroughs. The rising need for robustness in FR systems and investigated adversarial attack types on machine learning models and specifically Deepfool, a white-box untargeted adversarial evasion attack was discussed. The main objectives outlined by the research questions revolved around investigating how Deepfool could be altered to affect newer architectures in FR.

To answer the posed questions of this research, three FR models based on modern state-of-the art architectures were trained with the help of transfer learning. The models were subsequently used to evaluate the original Deepfool algorithm, along with a couple of Deepfool variants, which were modelled both in theory and practice to the best of our ability. We evaluated every model before the attack evaluation took place, with all three models showing little variations in performance given that they all stopped improving at around 96-97% validation accuracy, with a test error around 2-4% test error. Inception v3 seems to be performing slightly better in terms of prediction accuracy, where as Xception performs best in terms of prediction speed. All three Deepfool attacks were evaluated on the three models in terms of execution speed and adversarial robustness of the model.

The results show some interesting insights about the adversarial attacks, such as with the case of the Spiked Deepfool variant, which performs additional perturbations randomly at every iteration. Although evidence is shown to support that the models are slightly more robust against this variant, we see a decreased mean execution time, which suggests that the algorithm is able to find the minimal perturbation faster. Despite the faster execution however, it is unclear whether the additional perturbations cause a significant difference in the total perturbation added to the image, compared to total perturbations added by the original algorithm. The adversarial robustness value from the used formula by Moosavi-Dezfooli et al., should be indicative to a certain degree of the magnitude of total perturbations, yet the difference shown between adversarial robustness against the original from the Spiked variant is not that clear and should be further investigated [39]. The second variant, QGA Deepfool, even though it followed a more unique and advanced approach to enhancing Deepfool, seems to be performing worse than the original in terms of speed. By calculating additional gradient points to help in hyper-plane projections, the algorithm only seems to be increasing in complexity without any improvement, since this effect could be similarly achieved as iterations increase. In terms of robustness the QGA variant appears to perform similarly to the original. It could serve as good future project to optimise QGA's linear approximation of gradients capabilities.

The project was hindered by some limitations along the way, yet only some of these had a significant impact on the overall results and evaluation of the project. Namely, it has been observed in the evaluation of the three test models, that approximately 21-36 samples out 921 of the test were miss-classified. This number is almost negligible given the size of the whole set, yet the reason

behind the wrong predictions is some bad practices in face extraction and dataset cleaning. The evaluation showed that improper handling of samples with varying poses, ages and multiple faces in one image, contributed to bad predictions. The most important limitation however, was the execution and calculation of adversarial robustness tests, since testing single adversarial attacks was fast enough, but doing so for a set of 921 images resulted in hours of running executions for each model and attack we tested. Given that this project has been performed on a relatively mid-to-high end personal computer with decent DL capabilities, such tests appear to be better suited for dedicated and very well-equipped DL workstations which sadly this study could not get access to. Furthermore, some implementational struggles were present due to version incompatibilities of ML libraries and the use of outdated APIs.

Despite that we have managed to achieve our main set objectives with an acceptable degree of experimental limitations. Following the evaluation of the research questions, in the previous chapter, we can say that although the proposed Deepfool attack variants do not offer any groundbreaking insight on modern architectures, they were ultimately theorised and implemented successfully, albeit their results. Final notes on this study would point towards a shift of focus towards using other types of attacks on such architectures which are more recent and advanced such as Elastic Net [16]. Adding to this, using Deepfool does not appear to offer anything particular in the context of facial recognition specifically, hence this study could benefit from more focus on attacks that target processes of FR directly. Deepfool has only been chosen as the adversarial attack to evaluate because of its vast literature coverage and accessible material as well as impressive attacking speed.

All in all, it would be beneficial to further test the robustness of the three models after they have been fine-tuned with adversarial examples in the future. More regularisation options such as L2 kernel regularisation could also be used to reduce overfitting and promote more robustness in the model. A re-evaluation of the attack variants could also be done with comparisons to attacks other than Deepfool to examine success of more modern approaches in adversarial attacks on AI systems. Given the results of this research, it is believed that the Deepfool variants although not as effective, could be further improved to match the performance of other adversarial attack in the future.

Appendix A

Additional Material

A.1 Glossary

Glossary List	
Artificial intelligence	AI
Machine Learning	ML
Deep learning	DL
Facial Recognition	FR
Computer Vision	CV
Graphical Processing Unit	GPU

Table A.1: Some common terminologies and abbreviations which will be used in the project.

A.2 Evaluation and metric formulas

A.2.1 Classification report formulas

The metrics formulas used in the classification that will follow in this subsection, utilise the following definitions regarding classification predictions:

- **True Negative (TN):** scenario when a sample was **negative** but predicted as **negative**.
- **True Positive (TP):** scenario when a sample was **positive** but predicted as **positive**.
- **False Negative (FN):** scenario when a sample was **positive** but predicted as **negative**.
- **False Positive (FP):** scenario when a sample was **negative** but predicted as **positive**.

Below are the metric formulas used in the classification report[1]:

Precision :

$$\frac{TP}{(TP+FP)}$$

Recall :

$$\frac{TP}{(TP+FN)}$$

[1] https://peltarion.com/knowledge-center/documentation/evaluation-view/classification-loss-metrics/macro-f1-score

F1 Score :

$$\frac{2 \times (Recall \times Precision)}{(Recall + Precision)}$$

Support : Support does not have a specific formula.

Accuracy :

$$\frac{TP + TN}{\text{Total Input Images}}$$

Macro Average :

Macro Average Precision:

$$\frac{1}{N} \sum_{i=0}^{N} Precision_i$$

Macro Average Recall:

$$\frac{1}{N} \sum_{i=0}^{N} Recall_i$$

Macro Average F1-score:

$$\frac{1}{N} \sum_{i=0}^{N} F1score_i$$

, where i is the class/label index and N the number of classes/labels.

Weighted Average :

Weighted Average Precision:

$$\frac{1}{N} \sum_{i=0}^{N} Precision_i$$

Weighted Average Recall:

$$\frac{1}{N} \sum_{i=0}^{N} Recall_i$$

Weighted Average F1-score:

$$\frac{1}{N} \sum_{i=0}^{N} F1score_i$$

, where i is the class/label index and N the number of classes/labels.

Appendix B

Instruction Manual

This chapter serves as a guide to run and maintain the source code included with this project as well as the steps required to reproduce the results. As such by reading this chapter, the reader should be able to **install** required libraries and dependencies, **modify** the project code, **extend** it if they want and finally **trace** any bugs that come along the way.

B.1 Preparing environment and libraries

To be able to run the project code, the person handling the archive must follow a set of instructions to prepare a working python environment with the correct libraries and dependencies.

Hardware Dependencies: No specific hardware is needed. However it is **strongly encouraged** to use a machine with a **good GPU**, as it performs deep learning tasks faster.

Python version: 3.8.10

Download python here:

`https://www.python.org/downloads/`

Environment: Follow the link below to a guide explaining how to create a python **virtual environment**. Although not absolutely necessary, it is highly **recommended** as it will minimise dependency conflicts.

Venv Guide:

`https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/26/python-virtual-env/`

Jupyter: The project code is in the form of a Jupyter notebook. Hence, it is required to use Jupyter or Jupyter Lab. An alternative is to use cloud services such as Google Collab, however it has an upload storage limit, which will probably prohibit all dataset files from being uploaded.

Guide to setting up **Jupyter**:

`https://jupyter.org/install`

Libraries and Dependencies: All dependencies can be found in the *requirement.txt* file which is generated by using the command

pip freeze > requirements.txt

Doing so however, lists all libraries and their dependencies installed and not the main libraries only. Therefore, the *requirement.txt* file has been reduced to the main libraries required for the users convenience. Then the user is only required to run the following command to install these libraries:

pip install -r requirements.txt

Notes:

1. The libraries and dependencies mentioned before this item should also be installed (eg. jupyter).
2. Make sure to **activate** the virtual environment and be inside the main directory. Activating the environment can be done by:

.\venv\Scripts\activate

B.2 Project Directory structure

In this section the directory structure of the submitted items is presented. The use of each of these sub-directories and files is also explained. The list of files and directories is shown below.

- model_ckp /

Where model checkpoints prior to fine-tuning are saved.

- model_bkp /

Where plots and metrics are exported when evaluating the model.

- model_FT_ckp /

Where model checkpoints during fine-tuning are saved.

- saved_models /

Where the trained models are **stored** and must be **loaded** from. There is one model for each corresponding python notebook with that name. The Demo notebook also uses the Xception model due to speed. Refer to the next section for how to **download** the models.

- saved_models **/model_FT_final_EfficientNet.h5**
- saved_models **/model_FT_final_InceptionV3.h5**
- saved_models **/model_FT_final_Xception.h5**

- Deepfool_Model_final_Xception.ipynb

Jupyter notebook for the Xception model.

- Deepfool_Model_final_InceptionV3.ipynb

Jupyter notebook for the InceptionV3 model.

- Deepfool_Model_final_EfficientNet.ipynb

Jupyter notebook for the EfficientNet-B0 model.

- Demo_notebook.ipynb

Jupyter notebook for the Xception model.

- Adversarial attacks on modern deep facial recognition systems using Deepfool.pptx

Power Point Presentation file

- Code_notebook.ipynb

- **venv /** (when created by the user)

B.3 Data retrieval

Given that the submission guidelines have a small definite size limit, we have included some essential parts (numpy dataset arrays and trained models) of the source code in an cloud storage for users to access and download. This is a **mandatory** step, since there is no other way to get the custom dataset. For the users convenience, the dataset has been packaged in numpy files which can be loaded at specified checkpoints marked in the source code. More about these checkpoints will be mentioned in the next sections.

A **link** for for the whole **VGGFace2 dataset** can be found here:
`https://academictorrents.com/details/535113b8395832f09121bc53ac85d7bc8ef6fa5b`

*****Note***:** This is the only way to acquire the dataset currently as the official page link on VGG website's has been broken for a while. This link has been suggested by Weidi Xie from VGG through email as well as other members of online machine learning communities online.

The link for the **cloud storage** where the project code is stored can be accessed here:

`https://drive.google.com/drive/folders/1q3ytfAy6yb626CWMUnSl-ZQfUwen6URu?usp=sharing`

B.4 Created Models

This section lists the created models which were used for evaluation. The user can load these models by following the corresponding steps listed in the User Manual and the project code headings. It is important to note that even if the user loads these models and uses them to reproduce experiments, the results can be almost identical but at times not the same as the results presented in the evaluation. This happens because a lot of instances where the model is used for predictions uses *predict* and *forward pass* interchangeably as some attack operations demand so. In Tensorflow and Keras there are some discrepancies in how the two methods work.

- saved_models **/model_FT_final_efficientnet.h5**
- saved_models **/model_FT_final_InceptionV3.h5**
- saved_models **/model_FT_final_xception.h5**

B.5 User Manual

When the previous steps of creating an environment and installing required dependencies, in order to launch the project code the user needs to also activate the virtual environment and launch the notebook using either of the following commands, depending on what they installed.

Use only **one** of the following in a **terminal** with the **activated virtual environment**:

jupyter lab

or

jupyter notebook

A good tip to know that this process has been performed correctly is to check the python kernel which should be running and have the name of the python kernel or virtual environment (e.g **venv**).

B.5.1 Load prerequisite code blocks

The following code sections should be run in order for code blocks later to work. Most bugs that might appear should be arising from a code block that the user has neglected to execute. Thus make sure that the following are executed to be loaded in memory.

Section 1.1.0 - Execute to **import libraries** that are needed.

Section 1.2.1 - Load **Face extraction** and **rescaling** function

Section 1.2.2 - Load **dataset building** functions

Section 1.2.4 - Make sure you **download** the files from the **cloud storage** in order to load the numpy packaged dataset and skip rebuilding the dataset from scratch. Then proceed to execute the next code block in this section to train the label encoder and convert labels to one-hot arrays.

Section 1.2.5/1.2.6 - Here there are two options but it is recommended that you skip Section 1.2.5 that performs the train and test split, and instead just load the corresponding sets from the numpy files.

Section 1.2.7 - This step is very **crucial** especially if a training will be performed. Execute this block to **construct** a **Tensorflow dataset** with **batch size** 32, **preprocess** the data according to the model used, **shuffle** it and **prefetch** it for memory access optimisation.

B.5.2 How to train a model

For the user to train a model, the steps described in the previous section must first be executed.

Section 1.3.1 - Build the model by creating the layers on top of the transferred model in a single model instance.

Section 1.3.2 - Set **training** to **False** for the transferred layers, **compile** the model with an **optimiser** and **loss** function and train the new top for 50 epochs.

Section 1.3.3 - Execute this block after the initial training to **unfreeze layers** of the transferred model and **fine-tune** the whole model for 150 epochs with a new **optimiser** but same **loss** function.

Section 1.3.4 - Plot Accuracy/loss for training and fine-tuning. This is only possible if training is performed since the metrics are stored when the model is trained with *fit(...)*.

Note:** If the user wishes to train the model from scratch, then they must make sure that sufficient disk space is available since we are saving the model at each epoch. In the event that the user wishes to only save the model at the end then the following code must be edited and line 7 should be **removed** for both Section **1.3.2** and **1.3.3**.

```
1 history2 = model.fit(train_dataset,
2                       batch_size=batch_size,
3                       epochs=150,
4                       validation_data=test_dataset,
5                       verbose=1,
6                       callbacks=[early_stopping,
7                       ModelCheckpoint(filepath='model_FT_ckp/model_FT_at_ep{
    epoch}.h5'),
8                           ]
```

B.5.3 Evaluation with a loaded model.

Execute all of the designated code blocks mentioned in the prerequisite code blocks subsection, which lists steps prior to section 1.3.1, that marks the start of a model training process.

The following list enumerates the sections in order that have to be executed to evaluate a model. The ouput will be in the same format that was reported in the Evaluation chapter. If the user has already trained the model and does not wish to load from file, then **Section 2.1.0** should be skipped.

Section 2.1.0 - Load model from file. No training required. it is **recompiled** and **frozen** to put all layers in **inference mode**.

Section 2.2.0 - Convert test set back to **numpy** to perform all other evaluations.

Section 2.3.0 - Load **prediction functions**. These are needed for other functions used in the attacks section.

Section 2.4.1 - Generate predictions to calculate the **metrics** later.

Section 2.4.2 - Plot a **Confusion matrix**.

Section 2.4.3 - Generate **Classification report**.

Section 2.4.4 - Gather **miss-classified samples** and plot them.

B.5.4 Adversarial Attacks

The list that follows describes what each section does in regards to Deepfool adversarial attacks.

Section 3.1.1 - Pure Deepfool class.

Section 3.1.2 - Spiked Deepfool class.

Section 3.1.3 - QGA Deepfool.

Section 3.2.0 - Testing section for single random attacks on the test set. Code cells in this section can be executed to investigate simple attack instances.

Section 3.2.1 - Attack function. Performs a single attack specified by the Deepfool variant name on a random sample form the test set.

B.5.5 Adversarial Robustness Test

Finally this section is the last part of the evaluation presented in this document. The following list presents its items.

Section 3.3.1 - Adversarial Robustness Calculation function. This calculates the average execution time and adversarial robustness of a model on a specific attack variant.

Section 3.3.2 - Pure Deepfool adversarial robustness evaluation.

Section 3.3.3 - Spiked Deepfool adversarial robustness evaluation.

Section 3.3.4 - QGA Deepfool adversarial robustness evaluation.

B.5.6 TEST Section

This is a test section present in all notebooks, but is not used for anything meaningful to the main document. The user is free to add any code blocks here and execute code if they wish to investigate more with the source code.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.

[3] B. Amos, B. Ludwiczuk, and M. Satyanarayanan. Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[4] Arc. Deepfool - a simple and accurate method to fool deep neural networks., May 2019. URL `https://towardsdatascience.com/deepfool-a-simple-and-accurate-method-to-fool-deep-neural-networks-17e0d0910ac0`.

[5] T. Baltrusaitis, A. Zadeh, Y. C. Lim, and L.-P. Morency. Openface 2.0: Facial behavior analysis toolkit. In *2018 13th IEEE international conference on automatic face & gesture recognition (FG 2018)*, pages 59–66. IEEE, 2018.

[6] W. Bledsoe. The model method in facial recognition, panoramic research inc. *Palo Alto, CA, Rep. PRI*, 15, 1964.

[7] W. W. Bledsoe. Man-machine facial recognition. *Panoramic Research Inc., Palo Alto, CA*, 1966.

[8] W. W. Bledsoe and H. Chan. A man-machine facial recognition system—some preliminary results. *Panoramic Research, Inc, Palo Alto, California., Technical Report PRI A*, 19:1965, 1965.

[9] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[10] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017.

[11] K. Bálint. Uavs with biometric facial recognition capabilities in the combat against terrorism. In *2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY)*, pages 000185–000190, 2018. doi: 10.1109/SISY.2018.8524800.

[12] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman. Vggface2: A dataset for recognising faces across pose and age. corr abs/1710.08092 (2017). *arXiv preprint arXiv:1710.08092*, 2017.

[13] N. Carlini and D. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM workshop on artificial intelligence and*

security, pages 3–14, 2017.

[14] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. corr abs/1608.04644 (2016). *arXiv preprint arXiv:1608.04644*, 2016.

[15] L. R. Carlos-Roca, I. H. Torres, and C. F. Tena. Facial recognition application for border control. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2018.

[16] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh. Ead: elastic-net attacks to deep neural networks via adversarial examples. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[17] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[18] F. Chollet et al. Keras. `https://keras.io`, 2015.

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[20] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[21] A. Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA, 2019. ISBN 978-1-492-03264-9.

[22] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[23] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[24] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[25] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[26] M. Hussain, J. J. Bird, and D. R. Faria. A study on cnn transfer learning for image classification. In *UK Workshop on computational Intelligence*, pages 191–202. Springer, 2018.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[28] D. Kunda and M. Chishimba. A survey of android mobile phone authentication schemes. *Mobile Networks and Applications*, pages 1–9, 2018.

[29] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[30] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2, 1989.

[31] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[32] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[33] Y. LeCun et al. Generalization and network design strategies. *Connectionism in perspective*, 19:143–155, 1989.

[34] X. Liu, H. Yang, Z. Liu, L. Song, H. Li, and Y. Chen. Dpatch: An adversarial patch attack on object detectors. *arXiv preprint arXiv:1806.02299*, 2018.

[35] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *ICCV*, pages 3730–3738. IEEE Computer Society, 2015. ISBN 978-1-4673-8391-2. URL `http://dblp.uni-trier.de/db/conf/iccv/iccv2015.html#LiuLWT15`.

[36] Mathworks. Convolutional neural network, 2021. URL `https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html`.

[37] A. S. A. Mohamed, M. N. Ab Wahab, S. R. Krishnan, and D. B. L. Arasu. Facial recognition adaptation as biometric authentication for intelligent door locking system. In *International Visual Informatics Conference*, pages 257–267. Springer, 2019.

[38] I. Moisejevs. Evasion attacks on machine learning (or "adversarial examples"). *Towards Data Science, accessed July*, 21, 2019.

[39] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[40] F. E. Morgan, B. Boudreaux, A. J. Lohn, M. Ashby, C. Curriden, K. Klima, and D. Grossman. Military applications of artificial intelligence: ethical concerns in an uncertain world. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA SANTA MONICA United States, 2020.

[41] M.-I. Nicolae, M. Sinn, M. N. Tran, B. Buesser, A. Rawat, M. Wistuba, V. Zantedeschi, N. Baracaldo, B. Chen, H. Ludwig, et al. Adversarial robustness toolbox v1. 0.0. *arXiv preprint arXiv:1807.01069*, 2018.

[42] L. Ouannes, A. B. Khalifa, and N. E. B. Amara. Facial recognition in degraded conditions using local interest points. In *2020 17th International Multi-Conference on Systems, Signals & Devices (SSD)*, pages 404–409. IEEE, 2020.

[43] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010. doi: 10.1109/TKDE.2009.191.

[44] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[45] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition, 2015.

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[47] A. Polyakov. How to attack machine learning (evasion, poisoning, inference, trojans, backdoors). *Towards Data Science*, 2019.

[48] B. Rohrer. How do convolutional neural networks work?, 2016. URL `http://brohrer.github.io/how_convolutional_neural_networks_work.html`.

[49] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[50] S. I. Serengil and A. Ozpinar. Lightface: A hybrid deep face recognition framework. In *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pages 23–27. IEEE, 2020. doi: 10.1109/ASYU50717.2020.9259802.

[51] M. Shaha and M. Pawar. Transfer learning for image classification. In *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pages 656–660. IEEE, 2018.

[52] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human faces. *Josa a*, 4(3):519–524, 1987.

[54] L. J. Spreeuwers, A. J. Hendrikse, and K. Gerritsen. Evaluation of automatic face recognition for automatic border control on actual data recorded of travellers at schiphol airport. In *2012 BIOSIG-Proceedings of the International Conference of Biometrics Special Interest Group (BIOSIG)*, pages 1–6. IEEE, 2012.

[55] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. arxiv 2015. *arXiv preprint arXiv:1512.00567*, 1512, 2015.

[56] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning (2016). *arXiv preprint arXiv:1602.07261*, 2016.

[57] R. M. Taigman Y., Yang M. and W. L. Deepface: Closing the gap to human-level performance in face verification., 2014.

[58] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[59] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3 (1):71–86, 1991.

[60] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In *Proceedings. 1991 IEEE computer society conference on computer vision and pattern recognition*, pages 586–587. IEEE Computer Society, 1991.

[61] J. Vadillo, R. Santana, and J. A. Lozano. Exploring gaps in deepfool in search of more effective adversarial perturbations. *Machine Learning, Optimization, and Data Science*, page 215–227, 2020. doi: 10.1007/978-3-030-64580-9_18.

[62] F. Vakhshiteh, A. Nickabadi, and R. Ramachandra. Adversarial attacks against face recognition: A comprehensive study. *arXiv preprint arXiv:2007.11709*, 2020.

[63] Z. Wang, G. Wang, B. Huang, Z. Xiong, Q. Hong, H. Wu, P. Yi, K. Jiang, N. Wang, Y. Pei, et al. Masked face recognition dataset and application. *arXiv preprint arXiv:2003.09093*, 2020.

[64] Y. Wu and Q. Ji. Robust facial landmark detection under significant head poses and occlusion. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3658–3666, 2015.

[65] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503, 2016. doi: 10.1109/LSP.2016.2603342.