University of Bath

Master's Thesis

Optimising Facial Information Extraction and Processing using Convolutional Neural Networks

Author:
Eklavya Sarkar

Supervisor:
Dr. Wenbin Li

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Data Science

in the

Department of Computer Science

September 11, 2019

UNIVERSITY OF BATH

Abstract

Faculty of Science
Department of Computer Science

Master of Science in Data Science

Optimising Facial Information Extraction and Processing using Convolutional Neural Networks

Facial recognition and related technologies have impacted society in fundamental and often unpredictable ways. This thesis aims to provide a critical in-depth review of contemporary machine learning methods and an analysis of facial recognition, gender classification, and emotion detection problems, by implementing and evaluating models capable of solving these three tasks on personally collected data.

We first conduct an extensive literature review of the current techniques in use for solving such problems, define the scope and aims of this project, before evaluating the key data collection and pre-processing methodologies.

We then build multiple neural networks classes capable of training different types of models to solve all these tasks, and iteratively experiment with different architectures and methodologies to analyse and identify the parameters that are most optimal for extracting and processing facial information.

We train, tune, and compare the various models for each task, using built-in evaluative metrics. The subsequent results are analysed to determine the most efficient and optimal models for each task.

We then synthesise all the trained and most promising models into a single end-to-end prototype, capable of seamlessly executing all three tasks simultaneously. We then challenge the potential of this project by implementing a final model capable of processing real-time input data from a live web-cam.

Finally, the results of the project are evaluated in relation to its originally stated goals, and the strengths and weaknesses are discussed. A detailed roadmap of options for future work and further optimisation is then presented, before concluding with the overall learning points.

Acknowledgements

The journey leading up to the final submission of this thesis has been a formidable yet formative one, filled with hard work, lessons, triumphs, and immense learning to get to where I am now. The key to succeeding in this journey was a healthy mix of desire, ambition, inspiration, guidance, encouragement, and constructive criticism. I would like to take a moment to thank the people who contributed to completing my journey, and whose inputs were critical to the success of this project.

First and foremost, this project would not have gone beyond its conceptual stage without Dr. Wenbin Li, who agreed to an unarranged meeting in between his lectures, and was willing to not only supervise my project, but also help understand the scope and expectations of a Master's degree thesis. I thank him for giving me thoughtful and sensible guidance on efficiently managing my project, strongly encouraging me towards a PhD, and advising me on future career options.

Additionally, I would like to thank Dr. Willem Heijltjes for reviewing this thesis and being part of my educational journey. I am also grateful to Prof. Peter Hall for recommending Dr. Wenbin Li as my supervisor.

Furthermore, I would like to accord a special thanks to three particular individuals who directly and indirectly helped with my project. Chris Albion, Director of Data Science at Devoted Health, for his useful 'Machine Learning Flashcards' which were critical in developing my understanding of several technical concepts which were applied in this project. Rachel Tatman, data scientist at Kaggle Inc., for her inspiring and highly resourceful data science videos, as well as her willingness to help and correspond with users on Twitter. Mickaël Bressieux for taking the time to listen and discuss the technical challenges thrown up during the course of this project, and providing insightful feedback and ideas on how to overcome them.

Finally, I would like to extend my deep gratitude towards my family for their constant support throughout my educational journey. Specifically, I want to thank my father for recommending the University of Bath, and my mother for providing and maintaining a stress-free environment without which this thesis would not have been possible. Most of all, I thank my parents and sister for permitting use of their extensive archive of personal photographs, which formed the primary dataset for this project.

On a final note, I would like to thank my fellow Data Science coursemates for their kinship and steady network of support, the University of Bath's Computer Science Department for accepting me as their student, and finally the Student's Union Committee and the Salsa Society for making me feel very much at home.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AHE	**A**daptive **H**istogram **E**qualisation
AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etworks
AUC	**A**rea **U**nder the **C**urve
BGR	**B**lue **G**reen **R**ed
CNN	**C**onvolutional **N**eural **N**etwork
CLAHE	**C**ontrast **L**imited **A**daptive **H**istogram **E**qualisation
CLI	**C**ommand **L**ine **I**nterface
DL	**D**eep **L**earning
FER	**F**acial **E**motion **R**ecognition
GAN	**G**enerative **A**dversarial **N**etworks
GPU	**G**raphics **P**rocessing **U**nit
HSV	**H**ue **S**aturation **V**alue
ML	**M**achine **L**earning
MLP	**M**ulti **L**ayered **P**erceptron
NN	**N**earest **N**eighbour
LBP	**L**ocal **B**inary **P**attern
LDA	**L**inear **D**iscriminant **A**nalysis
PCA	**P**rincipal **C**omponent **A**nalysis
RBF	**R**adial **B**asis **F**unction
RGB	**R**ed **G**reen **B**lue
ROC	**R**eceiving **O**perating **C**characteristic
SGD	**S**tochastic **G**radient **D**escent
SVM	**S**upport **V**ector **M**achine

List of Models

TABLE 1: Authoritative list of models built for this project.

Task	Employed Code	Type
Facial Recognition	`mycnn.py`	Hold-out Softmax
Facial Recognition	`mycnn.py`	k-Fold Softmax
Gender Classification	`mycnn.py`	k-Fold Softmax
Gender Classification	`mycnn.py`	k-Fold Sigmoid
Emotion Detection	`mycnn.py`	Hold-out Softmax
Emotion Detection	`mycnn.py`	k-Fold Softmax
Emotion Detection	`mydeepcnn.py`	Hold-out Softmax
End-to-End	`RestoreModel.py`	Mixed
Real-Time	`LiveInput.py`	Mixed
Misc	`myutilitymethods.py`	Utility Methods
Misc	`convert_images_to_data.py`	Utility Script
Misc	`prepare_cropped_faces.py`	Utility Script

To my family,
the one at home and the one at university.

Chapter 1

Introduction

1.1 Background

Historically, the field of Artificial Intelligence (AI) has gone through several cycles of initial excitement, intense hype, optimism, and promises of revolution - dubbed as *AI summers*; only to be followed by periods of disappointments, aptly named *AI winters*, in which expectations failed to materialise, and government and research funds slowly moved on to other prospects [Chauvet (2018)].

We currently seem to be in a new AI summer phase, with several influential breakthroughs following the emergence of *Deep Learning*, a sub-field of the *Machine Learning* based on artificial neural networks. In the last decade, this has slowly surpassed the performance of existing statistical and classical Machine Learning techniques.

These new techniques improved, and became state-of-the-art, over the previous decade because of a few key factors, which were not necessarily related to conventional theoretical, mathematical, or algorithmic advancements. For example, the principle behind convolutional neural networks, a specific type of neural network now regularly used for computer vision and pattern recognition problems, was already well understood since 1989 [LeCun et al. (1989)]. Indeed, the current success of deep learning techniques stems from *engineering* and *technological* advancements, which loosened existing bottlenecks and allowed for a new period of research and development, which has led to many innovations we are currently witnessing.

A major breakthrough was improvement in hardware, which benefited both from Moore's law and the gaming industry. Companies such as NVIDIA had been continuously investing in the development of increasingly fast and 'parallelisable' graphical computing chips, used for rendering complex 3D scenes for personal computers. However, these developments happened to be equally useful for deep learning, or any other computationally intensive

tasks, which essentially consisted of matrix multiplications, since they are also highly parallelisable [Oh and Jung (2004)].

Another advancement was the increased availability of data and large datasets, which largely came from the growth of *storage* hardware capacities, as well as, the meteoric rise of crowd-sourced data through the Internet, often collected through modern media such as smartphones and social-media websites such as YouTube, Wikipedia, and Google, to name a few.

Finally, once these two elements had significantly improved, there was enough scope to experiment with new theoretical ideas, which led to methods which have now become the foundation of most deep learning techniques, such as better activation functions and modern optimisers. These advanced techniques, combined with networks of deeper layers, stronger hardware capacities, and larger datasets finally resulted in concrete breakthroughs in the mid-2010s. Since then, the AI field has naturally continued to evolve, and given rise to even more modern techniques, such as dropout units, batch normalisation, and generative adversarial networks, which have further contributed to the growth of AI [Minar and Naher (2018)].

Indeed, ever since large technology firms and industry leaders seriously invested in AI research, there has been a rash of AI-based start-ups, and inclusion of AI-based strategies in governmental planning through national think-tanks. The exponential growth in this field has - and continues to have - an impact on society through means of various products, such as autonomous vehicles, hyper-realistic ageing apps, chat-bots capable of making bookings, programs which can beat humans at any game, and videos of ourselves saying something we never did [Benaich and Hogarth (2019)].

Notably, the field of computer vision and pattern recognition has seen remarkable progress because of developments in deep learning, especially the breakthrough of convolutional neural networks. Perhaps the most significant task in this field is *human face recognition*, which has a number of potential applications, with deep social and political implications. Many ethical and privacy related debates revolve around the implementation and use of facial recognition technologies by large corporations or governments, through the means of new products or biometrics [Woodhouse et al. (2019)]. Concerns have grown over the surveillance of citizens by the State, using facial recognition technologies. Furthermore, in some countries, police have started wide-scale implementation using facial recognition models to apprehend suspects, a method which has been met with mixed results and significant public outcry [Winston et al. (2019)]. In the 2019 Hong Kong

demonstrations, many protesters made a point of spraying surveillance cameras, bringing down facial recognition towers, and using umbrellas or masks to conceal and protect their identities, to make a statement against such State-sponsored automation [Mozur and Lin (2019)].

Some technologies have progressed further than relatively simple facial recognition, and now attempt to detect the specific *emotions* present on a human face. As these methods continue to become more sophisticated over time, their impact will be continuously felt in our personal and professional lives. Given the pace of these developments, it is imperative to further explore deep learning research and its potential future applications. To this end, we come up with a problem which explores these issues in order to provide a study and in-depth analysis of facial recognition and related technologies, explained in the next section.

1.2 Problem

In the modern era of smartphones, most users today have a large dataset of images, many of which contain human faces. This project attempts to specifically make use of this abundant personal data, and combine it with modern deep learning techniques to demonstrate the power and potential applications of such technologies. Specifically, the aim of this thesis will be to solve the following main tasks: face detection, facial recognition, gender classification, and emotion detection, each of which shall be first thoroughly investigated in the form of a detailed literature review.

The proposed problem's aims and objectives can be broken down into essential and desirable features, as given below. The final implemented models should at least be able to:

- Detect a face in a given image.
- Crop the image down to the detected face.
- Recognise if a given face is either a specific individual or not.
- Classify the gender of a given face.
- Classify the expression of a given face according to whether it is happy or not happy.

Ideally, the final product should also be able to:

- Detect multiple faces in a given image.
- Recognise and classify the individual for a given face, based on the classes in the training set.

- Have the ability to take in several images, and output albums of images sorted according to each individual.
- Classify the expression of a given face according to 6 designated categories: happy, sad, angry, disgusted, fearful, surprised.
- Construct a pipeline which is able to link together the tasks of all the models and produce the 'before' and 'after' photographs.

The essential idea behind this conception of the project is to take advantage of the improvements which have occurred over the last decade, namely the widespread availability of high-performance computing hardware and personal data, and combine it with emerging deep learning techniques, in order to investigate and demonstrate how information from a human face can be efficiently extracted and processed, to a high degree of accuracy. It would also demonstrate how a cutting-edge AI product can today be independently developed by an ordinary researcher, armed with just basic theoretical knowledge and without the resources of a large corporation.

In the next chapter 2, we investigate and analyse the evolution of facial recognition and related technologies through the means of an in-depth literature review.

Chapter 2

Literature Review

2.1 Face Detection

2.1.1 Overview

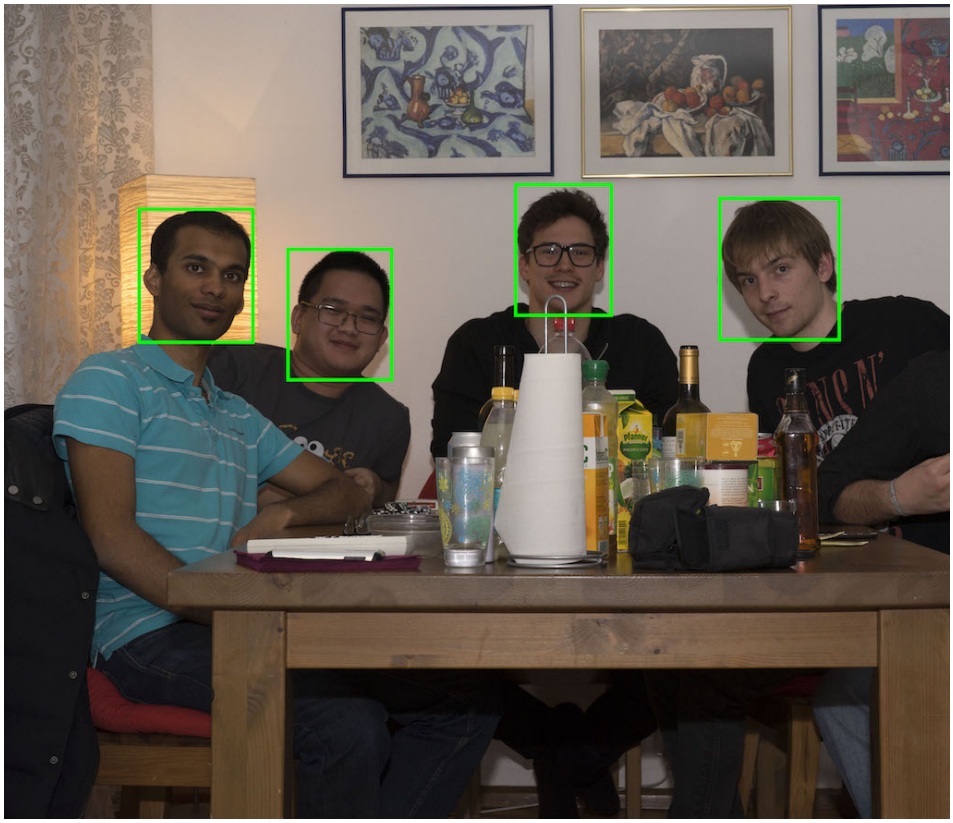

FIGURE 2.1: Sample face detection on a given image.

Face detection is the first and foremost objective of our formulated project, and is the foundational keystone for all facial tasks, including face clustering and emotion classification. Face detection, in our case also equivalent to face

This chapter substantially draws from and builds upon an essay written and submitted to Turnitin by self (Eklavya Sarkar) as an academic assignment in March 2019 for Module 'CM50175 Research Project Preparation'.

localisation, is the process of detecting a face on a given visual medium, such as a photograph, painting, or drawing. More specifically, it implies being able to detect the regions of a picture in which a face is present, regardless of its orientation, lighting conditions, or other issues.

In the case of videos, face detection could theoretically be done both in post-recording or real-time. It would require not only localising the face at a given instance of the video, but also continuing to track its location as it moves through a sequence of frames. This process is called *motion tracking*, and will not be covered in this project, as our goal and training data is focused static images.

Over the years, the techniques used to solve this problem have evolved to become more sophisticated in their approach, and have thus been able to overcome some of the challenges. Section 2.1.2 of this chapter provides an overview of the challenges in face detection which are common to all approaches. Section 2.1.3 then gives a few ideas of applications based on this process, before sections 2.1.4, 2.1.5, 2.1.6 explain the evolution of the techniques. Finally, section 2.1.7 summarises and concludes the chapter by giving future directions of this field.

2.1.2 Challenges

The task of face detection comes with a subset of its own unique problems. Depending on the approach taken, some of the issues described below may be circumvented, however, many of these are inevitable and common problems regardless of utilised technique.

The most obvious one would be the *obstruction*, or *occlusion*, of a face. As the model can only view an image as a static 2D one, it has no way of knowing if a face has been partially or entirely obscured by something else, and thus cannot detect it. Another evident issue is a lack of resolution in pictures: if the quality of an image is poor, then it could be harder to summarise a group of pixels as a 'face'.

The most important problem is perhaps the *lighting conditions* of a given photograph. The brightness, intensity, and distribution of the light can have huge effects on a model's ability to correctly detect a face.

Another factor - one particularly challenging to solve - is the person's *pose* and *orientation*. A picture can have a person in a *frontal* or *profile* pose, and the subject's head can be orientated at a number of varying angles.

The final set of problems relate directly to the person's face itself. Some models are unable to detect faces containing facial hair. Others only work well on humans with light-coloured skin tones. The actual expression of the photographed person can also be an issue - if it is particularly unusual, it could distort the key facial features required by some methods.

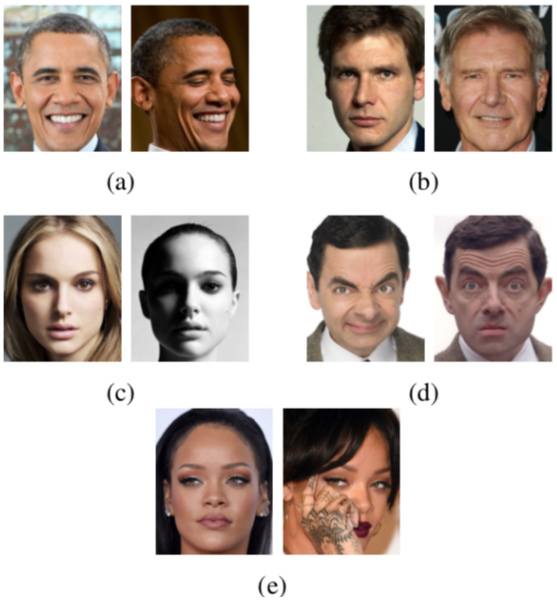

FIGURE 2.2: A variety of known faces under different illuminations, angles, and ages. Taken from Saez-Trigueros, Meng and Hartnett (2018).

2.1.3 Applications

2.1.4 Traditional Methods

The techniques used for face detection can classified in different categories based on the chosen parameters. Yang, Kriegman and Ahuja (2002) summarise them into four broad categories: knowledge-based, feature-based, template-matching, and appearance-based approaches.

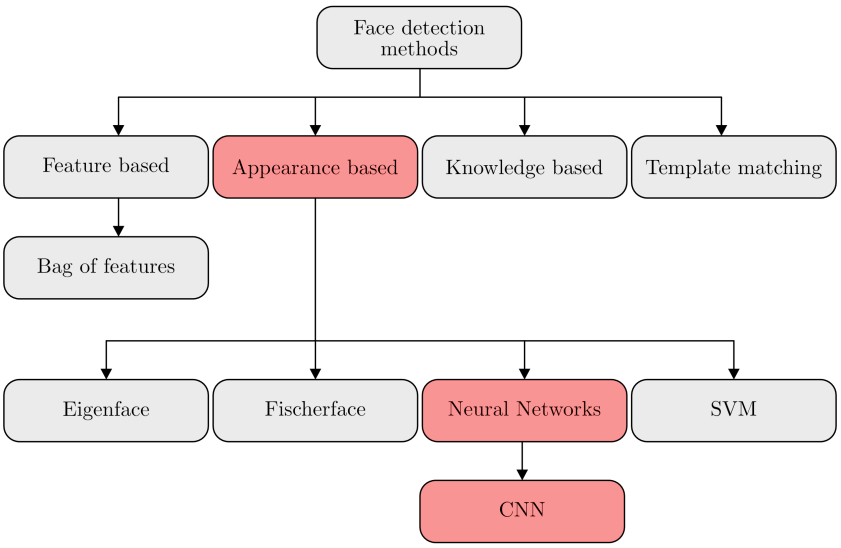

FIGURE 2.3: Set of techniques for face detection.

Knowledge-based Techniques

The knowledge-based methods were the first approaches undertaken for this task, as they are the easiest to define. They are, in essence, rules based on our knowledge of human faces and their common characteristics. For example, a human face must contain a nose, mouth, symmetric eyes within a certain distance. This allows a model to detect and extract these *facial features*, and then potentially classify the region in question as a face. The enactment of the defined rule has a big impact on the faces detected. A lenient set of rules would result in too many false positives, where as a strict set would likely only detect frontal, well illuminated, unobscured faces in uncluttered scenes.

An example such an implementation was done by, Yang and Huang (1994) who developed a hierarchical system which used a set of three rules, one for a new layer of abstraction. At the highest level, they simply scanned an entire image using a window looking for potential faces based on a general rule of what a face looks like. At the next level, they performed *local histogram equalisation* on the selected candidates, a processing method which equalises the contrast of an image, and then *edge detection*, which simply detects the edges. The rules at this level can thus relate to the lighting and histogram distribution of the image, e.g. 'the upper round part of a face has a uniform intensity'. Finally, the remaining candidates go through a last round of rules screening, relating directly to the facial features, such as their mouth and nose. Although a model using this technique was able to detect 50 images which contained faces, out of a given test set of 60 pictures, 28 of them unfortunately were false positives.

Feature-based Techniques

Some approaches had a different fundamental outlook - they focused instead on the underlying assumption that there exists features or properties of faces which are *invariant*, as humans can effortlessly detect them regardless of the given poses and lighting conditions. These type of bottom-up methods start by detecting and extracting facial features, and then describing their relationships by developing a statistical model, which also determines the presence of a face. These type of models are often easy to implement, however, they are naturally very dependent on the quality of the image, and can be quickly crippled if the edges are too imperceptible or strong.

One of the significant papers in feature-based techniques, and face detection history in general, for a long time was by Viola and Jones (2001). It proved to be a landmark paper, and a turning point due to it robustness and extreme speed, and eventually became a foundational based technique, upon which many more incremental methods were proposed. It provided a machine learning approach for general object detection with very high processing and detection speeds. This was due to 3 key elements: a new image representation (named 'integral image'), a learning algorithm based on AdaBoost, and cascade-method which can combine increasingly complex classifiers. It is especially this last part which provides the dramatic increase in speed, approximately 15 times faster than any previous approach at the time.

Even to date, many modern tools provide face detection based on the groundbreaking work of Viola and Jones (2001). For example, OpenCV's current implementation of face detection is based on Haar's algorithm, and can even be implemented to work in real-time.

Template-matching Techniques

Another popular approach is template-matching, in which a standard face's pattern is manually defined or parametrised by a function. We can then use a model to correlate a given face's values to the standard one, and determine the presence of a face according to them. While these models are relatively easy to implement, they have trouble with different poses, shapes and scales. As such, there are two types of templates which could alleviate the problem.

Appearance-based Techniques

Unlike the template-matching techniques which rely on given *predefined* templates, appearance-based methods *learn* them during training. It is

these approaches which are currently having a breakthrough, and they are very much based on principles of statistical analysis and machine learning. Indeed, they are able to determine the key characteristics of faces in the form of distribution models or discriminant functions, and then use them to correctly identify faces themselves. These methods tend to be more computationally extensive, which is why *dimensionality reduction*, a process of reducing the number of random variables, is often executed on the training dataset.

Many of the appearance-based models work on a probabilistic framework - a given variable can be classified as belong to the 'faces' class or the 'non-faces' class. If given a space of images, it will contain individual vectors representing face or non-face images. The region occupied by faces vectors is usually only a small subspace of the total image space. We can compute the k-dimension subspace such that the projection of the data points onto the subspace has the largest variance among all n-dim subspaces. This captures the essential appearance characteristics of faces.

Eigenfaces

Turk and Pentland (1991) pioneered a key algorithm in face detection based on *Principal Component Analysis* (PCA), a dimension reduction technique involving identifying the principal variables.

If we assume that most faces lie on low-dimension subspaces determined by the first k directions of maximum variance, we can essentially use PCA to determine the vectors that span that subspace. We can represent all the given faces in the dataset as the linear combinations of eigenfaces, which are the principal components.

As explained by Krishna (2018), the strongest advantage of this approach is that it requires no explicit knowledge about the faces, or their expressions, during training, as there is no attempt at preserving class distinctions. Furthermore, it is also a non-iterative and globally optimal solution.

On the other hand, however, there are a few requirements for the testing algorithm to adequately work. First of all, all the faces must be aligned to the centre frame to avoid noisy results. Secondly, the images must all be the same size. The algorithm is also susceptible to a certain degree to the angle of the face, i.e. the pose. It is also important to note, since the method is knowledge free, PCA doesn't take into account the labels associated with the faces, and thus it could map dissimilar faces in the same regional subspace, making it tough for the classifiers to distinguish between

them.

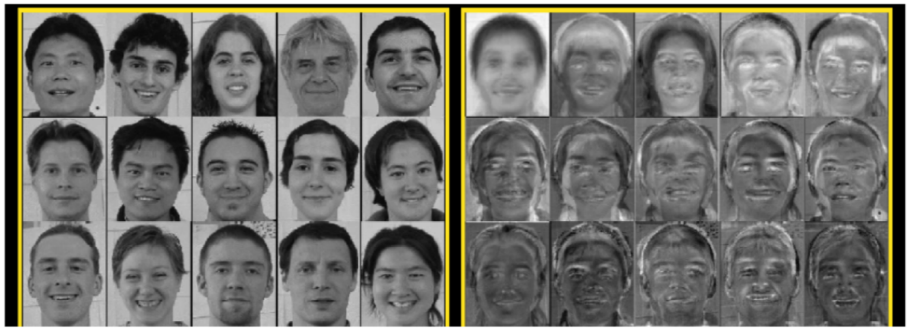

FIGURE 2.4: Faces (left) and eigenfaces (right). Taken from Krishna (2018).

Fisherface

Fisherface technique is an enhancement on Eigenfaces, as it uses Fisher's Linear Discriminant Analysis (LDA) instead to PCA for dimensionality reduction. Since PCA doesn't take the labels of classes into account, LDA was proposed as an alternative which is optimal for classification, as opposed to reconstruction. LDA tries find a projection which maximises the mean distance between classes *and* minimises the scatter within a class, where as PCA simply looks for the highest variance within a class. This allows for an improved overall class discrimination.

In a survey by Belhumeur, Hespanha and Kriegman (1996), the results by Eigenface and Fisherface were compared in depth on a dataset of 160 images of 10 distinct people, under 10 different lighting conditions or expressions. They arrived at the conclusion that the Fisherface approach appears to be better at handling variation in lighting and expressions, and has significantly lower error rates than Eigenface. The latter does improve by a margin if the three largest components are removed, but does not yield better results than Fisherface.

Support Vector Machines

Support Vector Machines (SVMs) are popular in machine learning for supervised and classification problems, and can also be applied here for face localisation problems. The are linear binary classifiers which attempt to maximise the margin between the *decision hyperplane*, i.e. the classification 'line', and the examples in the training set. As explained by Yang, Kriegman and Ahuja (2002), the 'best' hyperplane is given as a weighted combination of a *subset* of the training vectors, called support vectors, and can be estimated by solving a bounded linear quadratic programming problem.

This was first attempted by Osuna, Freund and Girosit (1997), who presented a novel and stable decomposition of this algorithm, and tested it on large datasets. They were able to optimise their algorithm, and make it computationally efficient enough to show that SVMs yielded results comparable to the then state-of-the-art models.

2.1.5 Turning Points

Neural networks, which have been applied to most pattern recognition problems by now, have successfully gotten good results for face detection. Similar to knowledge-based techniques, neural networks can use a hierarchical system to get an enhanced overall performance.

Such a system was first proposed by Agui et al. (1992), who used two layers. The first consisted of two parallel sub-networks which filtered the intensity values of a given image. The second one would then take the outputs of the previous layer and the extracted feature values, such as the standard deviation of the pixel values in the input pattern, as its own inputs. The existence of a face in the input area would be revealed through the second layer's output. They observed through experimentation that this method would produce good results if the test images contained faces of the same dimensions.

Propp, Samal and Ashok (1992) worked on a similar project, but with a higher number of layers and inputs. They had a total of four layers with 1024 input units, 256 for the first layer, 8 for the second, and finally 2 output units.

Burel and Carel (1994) implemented a network which conducted dimensionality reduction on the training set of boht faces and non faces, using an algorithm developed by Kohonen (1989). They used a multi-layered perceptron (MLP) to train through these examples for face classification, and detected them by scanning all the images are different resolutions. They also performed normalisation of the lighting conditions, and standardisation of the image size, before using the MLP to classify the scanning window's content.

Some worked on auto-associative neural networks, with five layers, which is able to compute a non-linear PCA. One of these can be employed to localise frontal faces and another for detecting angled ones, as was done by Feraud et al. (2001).

Neural networks designed in a probabilistic manner as named probabilistic decision-based neural network (PDBNN), and resemble radial basis function (RBF) networks. Lin, Kung and Lin (1997) proposed such a system which extracted feature vectors solely on the basis of the intensity and edge information of the central facial region (containing the mouth, nose, eyes and eyebrows). The two selected feature vectors are individually input into two different PDBNNs, whose output are merged together to determine the classification of the system. Their results were comparable to other leading neural facial detection networks.

A landmark paper using neural networks was presented by Rowley, Baluja and Kanade (1996), who also used a multi-layered perceptron. Unlike some of their contemporaries who tried to find an optimal discriminant function to classify faces and non-faces using distance measures, Rowley, Baluja and Kanade (1996) tried to map the relationship between these two entities's patterns using their image's intensities and spatial relationship of pixels. As explained by Yang, Kriegman and Ahuja (2002), there are two major components used in their system, one is a multiple neural network, used to detect face patterns, and the other a decision making module, used to render the final decision from numerous detection results. The model receives a region of a given image as input, and outputs a value between -1 to 1, indicating the certainty to a non-face or a face respectively. The network is applied to all locations in the image. If a face happened to be bigger than the size of the input region, it would go by undetected, which is why an input image is repeatedly sub-sampled and the network is applied at each scale.

However, this system could only detect upright and frontal faces, until they published another paper with added methods to detect angled poses thanks to a *router network*. This essentially processed the given input region to determine the angle of a possible face, then rotated the regional window to a canonical orientation, which is then once again presented to the neural network. Although the detection rate of with the extended methodology was lower than the original one, it still had very few false positives at 76.9%.

2.1.6 State of the Art

In more recent papers, however, deep learning approaches have been undertaken to tackle this problem with fruitful results. Farfade, Saberian and Li (2015) called this model 'Deep Dense Face Detector' (DDFD), which did not require any pose or landmark annotation, as it was able to detect faces in a wide range of orientations using a *single model*. This was fundamentally due to the use of *convolutional neural networks* for feature extraction,

and modern GPU computational power. Another key idea of this model was to not use any additional methods such as segmentation, bounding-box regression, or SVM classifiers, as it significantly increases the computing complexity.

By analysing detection confidence scores, Farfade, Saberian and Li (2015) observed a correlation between 'the distribution of positive examples in the training set and the confidence scores of the proposed detector', suggesting the results could still be improved by using better sampling strategies and sophisticated data augmentation techniques.

Their method showed very positive results for rotated, occulted and multiple faces. It was comparable to R-CNN or cascade-based other face detection methods which are developed significantly for multi-face detection.

2.1.7 Conclusion and Future Directions

In this chapter, we reviewed the challenges and applications of face detection, before going through all the types of approaches used through the times to solve this problem, such as feature-based, appearance-based, template-matching, and so on. We observed the slow turning point that came with implementation of neural networks, before delving into the current progress made by deep learning.

Overall, face detection, while an interesting problem to solve in and of itself, is still is more of a pathway to one of greatest and most interesting problems in computer vision today, face recognition and facial emotion detection, and those sub-fields that have a lot more variables than the binary problem of classifying a region into a face or non-face.

2.2 Emotion Classification

2.2.1 Overview

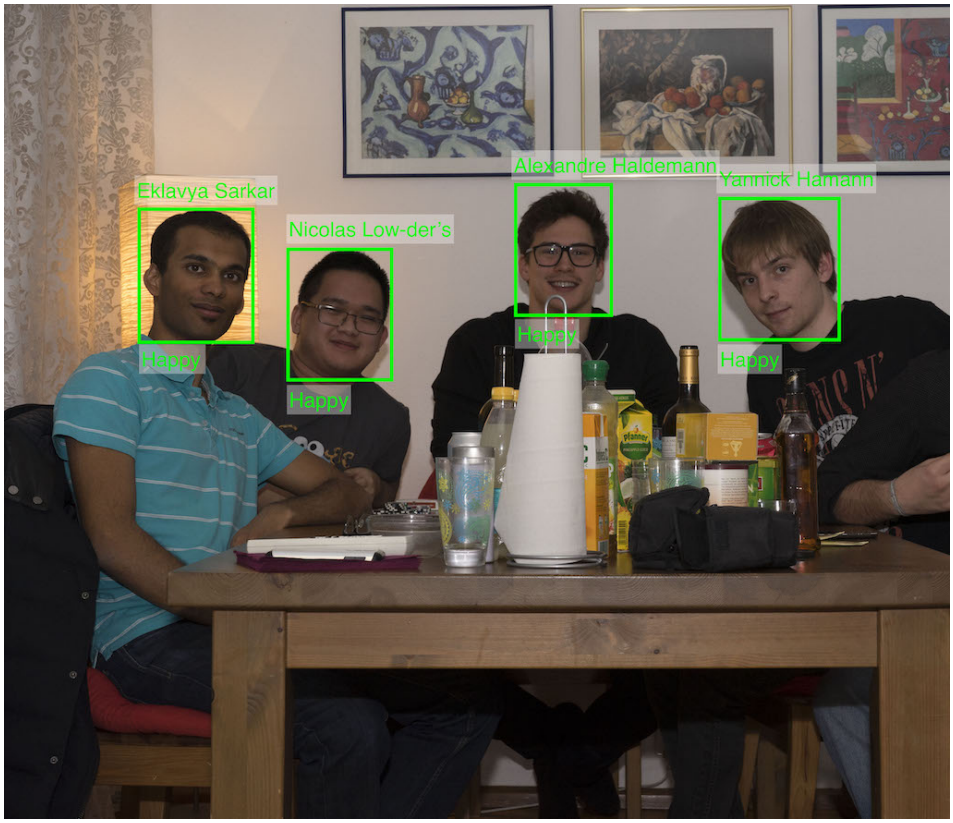

FIGURE 2.5: Sample facial expression classification.

Facial emotion, or expression, detection and classification is the final objective of our planned project - it is the process of determining the expression represented on a given face on an image. From the point of view of analysing a picture, it is arguably the final information which could gathered from a human face, after detecting and classifying it. It is an interesting and challenging task with a lot of value placed on it by the computer vision community.

Expressions epitomise social and non-verbal communications between humans, and automatic recognition of such emotions represents a big gateway to a number of novel applications and potential recommendation systems. It would also represent a big step not only in making static images more 'readable', but adding a third dimension of feelings/emotions to a picture. Expressions are the 'visible and mutative manifestation of human cognitive activity and pyschopthology' (Ou, 2012).

Which is why, there has been active research and significant progress made

in this sub-field. Methods now exist which are able to recognise a set of predefined 'basic' expressions, which convey universal emotions, usually summarised by happiness, sadness, fear, disgust, surprise and anger. However, like the previous two tasks, progress is being made to detect these classes under varying lighting conditions and poses. We first explain the numerous challenges unique to this task in section 2.2.2, before going through the potential applications of such a system in section 2.2.3. Section 2.2.4, 2.2.5, 2.2.6 again provide an overview of the evolution of the method used to solve this problem over time, ranging from the classical techniques to the current state-of-the-art approach. Finally, section 2.2.7 concludes by providing a summary and outlining the future directions of this problem.

2.2.2 Challenges

Facial expression recognition's (FER) main objective is to be able to automatically, reliably and efficaciously convey information about a person's expression. As these are only represented by the person's facial features, the most important factor, and biggest challenge, in this task is performing a very high level of feature extraction. As explained by Ou (2012), if inadequate features are extracted from an input, even the best classifier would fail to achieve accurate recognition. Therefore, the main difference between models tends to be in how they extract facial features, and then how they categorise them.

2.2.3 Applications

The applications for FER are numerous and can be applied to a lot of different context. As with face recognition, the most obvious example would be in the context of security systems, which could use them, along with other object detection, to tell if a person is in danger or not. For example, automated surveillance systems could use security camera feed, and when detecting both a knife and a person being fearful, can alert authorities. Drones could also use such technologies in a number of ways.

Another big sector of applications lie in recommendation systems, such as Spotify or Netflix. Currently, most of these rely on previously mined data on a user's habits, clicks, preferences and personal information to suggest content, but they have no definite way of predicting or telling what a user may be in the mood for *in the present*. A model which could take into account the person's current feelings by reading their mood and output highly accurate recommendations might be the closest we'd get to non-invasive systems until brain-computer interfaces. This could represent a new series of 'smart' devices capable of interacting with agents based on their mood.

Furthermore, emotions could also be used to narrow down searches in photos, along with time and geo-location information, or simply sorting photos according to basic emotions. Any algorithmic improvement for large technological firms can represent millions of additional revenue, and is thus a very competitive field of research.

Facial emotion recognition could also be used in health sectors, where the mental state of a patient is of importance. Automatic counselling systems could also be looked into. The applications are only limited by one's imagination.

2.2.4 Traditional Methods

The overall pipeline for facial emotion recognition is similar across methods. It involves a model taking in an image, computing some pre-processing required to detect faces, after which the facial features, such as eyes, mouth, nose, etc., are extracted and used as inputs in the final classifier which determines the expression of the individual in question based on its classification method. This is usually either a Support Vector Machine (SVM) or a Nearest Neighbour (NN). Is it the feature extraction step which varies most from method to method, and the following gives a quick overview of the most popular approaches used, as explained by Kumari, Rajesh and Pooja (2015).

Local Binary Pattern (LBP) works by labelling each pixel in a region, defined by a chosen radius, with a decimal value calculated by thresholding each pixel value by their central value. Local Gradient Code, on the other hand, tries to work out the relationship of *neighbouring* pixels. Local Direction Pattern (Jabid, Hasanul Kabir and Chae, 2010) attempts to get an improved performance despite variation in the lighting conditions, by using masks which are convolved on regions to get mask values, which are in turn ranked and assigned values. Finally, histogram of gradient orientations uses the X and Y gradients of an image, which calculated using a gradient filter, to calculate the corresponding magnitude and angle orientations. These are then split up into 'bins', and the image itself is divided into 'cells'. The magnitude is repeatedly separated into bins corresponding to the angular section to which it fell, and the obtained bin values are normalised to enhance the contrast.

The results presented by Kumari, Rajesh and Pooja (2015) showed that Local Gradient Code performed the best out of these methods.

The classification algorithm itself, used right at the end of the overall

pipeline, of a facial expression into one of the six universal aforementioned emotions can go awry if given any type of minor or major deformations. The facial landmark algorithms also have a few popular methods, namely, holistic, local, and regression-based.

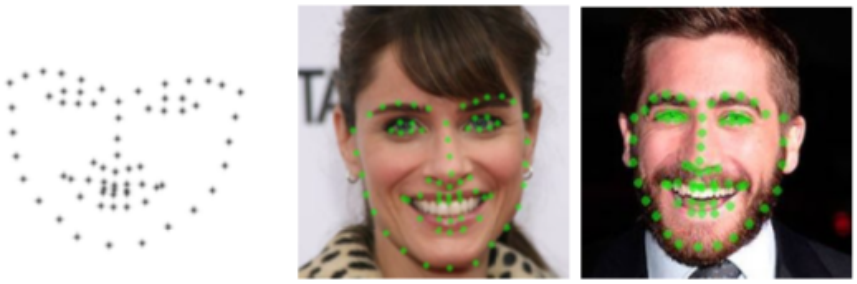

FIGURE 2.6: Facial landmarks defining the face shape, and the corresponding sample images. Taken from Saez-Trigueros, Meng and Hartnett (2018).

Holistic Methods

Holistic methods leverage and model the whole face and global facial shape patterns. The original holistic model was a statistical one called the 'Active Appearance Model' (AAM), developed by Cootes, Edwards and Taylor (2001). It uses methods such as 'Procrustes Analysis', PCA, and image wrapping. Most of the holistic methods since then have focused on improved the 'fitting algorithm', using two different approaches, called analytic fitting methods and learning-based fitting methods. The latter proved to be faster but less accurate. Holistic methods can range from slow to fast in terms of speed, and from poor to good in terms of performance (Wu and Ji, 2018).

Constrained Local Models

Constrained local models rely on the independent local facial appearance information and global facial shape patterns to infer landmark locations. The former is both easier to capture and more robust against illumination and occlusion, especially when compared to holistic methods. CMLs can be defined as both deterministic or probabilistic. The face shape model looks at the various facial landmarks and maps the spatial relationships between them, which constraint and refine the landmark location search. These methods generally perform better than holistic ones.

Regression-based Methods

Regression based methods use holistic or local appearance information, potentially embedded with with global facial shape patterns for joint landmark

detection. These are completely different from holistic or constrained local models, as they do not explicitly build any global face shape model, although the face shape constraints may be implicitly embedded (Wu and Ji, 2018). These can be categorised into direct regression methods, cascaded regression methods and deep-learning regression methods.

Direct regression models map the image to facial landmark locations in a single step, which can further be classified into local or global approaches. The former samples different areas of the face region, and builds regression models to predict the displacement vectors, and then added to the local current patch location to produce all landmark locations jointly. Combining the predictions from multiple samples patches gives the final landmark locations (Wu and Ji, 2018).

Global approaches map global facial images to landmarks directly, as they convey more information for landmark localisation. This technique however has proven to be more difficult to learn, as the global facial appearance has substantial variance and can be prone to obstruction.

2.2.5 Turning Points

Unlike direct regression models, which are able to predict locations in a single step, **cascaded regression methods** start from an initial guess of the facial landmark location, usually the mean face, before incrementally updating the locations across stages with various regression functions learnt from different stages (Wu and Ji, 2018). Regressions models are used to learn the relationship between shape indexed image appearances. The model learns and updates the training data at every stage, before being sequentially applied to update the shapes across iterations during testing.

There have been attempts at modified versions of cascaded regression methods as well. Instead of having a structure in which the layers are dependent on the former ones, a parallel learning method was developed by Asthana et al. (2014). This resulted in faster training, as it allowed for incremental updates to the model parameters at each level by adding a few more training samples.

Generally speaking, the cascade based regression methods have proven to be more effective than the direct ones, as they follow a 'coarse-to-fine' methodology, going from focusing on large to fine variations with their regression functions. However, there are a few issues with this approach, namely, the lack of clarity on how to generate the initial landmark locations. While one can use the mean face, it is does not work effectively for images with large

head poses. Furthermore, cascaded methods apply only a fixed number of predictions, and there is no concrete way of assessing the quality of the landmark prediction or adapting it for testing images.

2.2.6 State of the Art

In recent times, there has been an overall shift towards using **deep learning** tools for solving computer vision problems, and facial expression recognition and landmark detection is no exception. Specifically, convolutional neural networks (CNNs) have cemented themselves as the current leader preferred choice of model, as most of them follow the global direct or cascaded regression methodology. However, the exact architecture of the CNN models vary from paper to paper, and behave differently. Some state-of-the-art models involve using an ensemble of CNNs to yield the highest performance, where predictions are integrated via some form of averaging.

The majority of models using CNNs used the same pre-processing methods described earlier, namely, involving face detection, illumination correction, histogram equalisation, facial landmark extraction, and linear plane fitting.

The architectures of the CNN models usually involve different permutations of convolutional, pooling, response-normalisation, inception, and fully connected layers. The choices also vary substantially in terms of the depth and number of parameters. Most architectures have surprisingly been shallow in relation to those in related fields (Pramerdorfer and Kampel, 2016). It is important to note that deeper networks do not necessarily have a higher number of parameters. Furthermore, Khorrami, Paine and Huang (2015) in fact demonstrated that a depth of five layers is sufficient enough to discriminate between high-level features for FER.

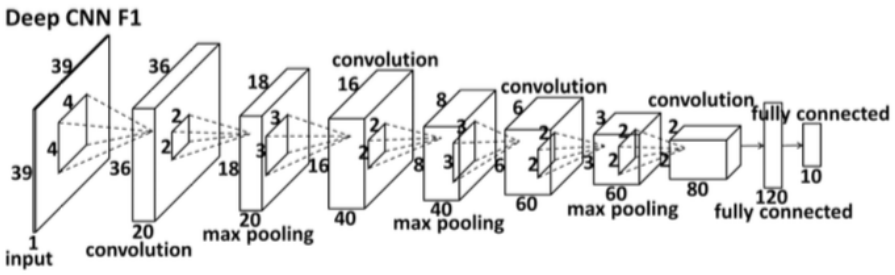

FIGURE 2.7: Sample example of a deep CNN. Taken from Wu and Ji (2018).

Some papers showed that having additional training dataset than just the FER2013 improves performance as it reduces bias (Zhang et al., 2015), especially since the FER2013 is relatively small. Both data augmentation

and ensemble voting have shown to improve generalised performance. Furthermore, according to Pramerdorfer and Kampel (2016), the three highest performing methods use face registration, revealing that this it beneficial even under sub-optimal conditions. However, they were able to show that an ensemble of shallow CNNs, can outperform the three best performing methods without using face registration or data augmentation.

The current bottleneck for models in FER is in fact the lack of a large labelled public training dataset. While face recognition for example has dataset with hundred of thousands or millions of images, facial expression datasets are much more limited, as each image has to be laboriously manually labelled. The FER2013 dataset, one of the largest ones available, only contains around 35 thousand such images.

2.2.7 Conclusion and Future Directions

In this chapter, we reviewed the challenges and applications of facial emotion recognition, and an overview of the approaches taken through history to solve this problem. Specifically, we went in depth in reviewing the traditional facial landmark detection methods, such as holistic, constrained local, and regression based techniques. We also delved into the current deep learning techniques, and the pipelines and architecture of the neural network models used in such models, and how these attempt to circumvent the problems relating to illumination and occlusion. We saw that convolutional neural networks have made significant progress in this field, but that bottlenecks still exist due to a lack of training data, and a deeper and more complex model isn't necessarily currently giving higher results than a shallow model.

Despite the progress made in recent times, these are still mostly trained, applied and measured against prepared facial datasets. 'In-the-wild' emotion recognition is still very challenging, although the problems are very much the same as in the prepared datasets. A lack of a generalised solution is likely contingent on the availability of a more thorough dataset, which might see a breakthrough in the coming years through the development of state-of-the-art Generative Adversarial Networks (GANs), which are complex models capable creating 'fake' images of faces. Nonetheless, the journey to attain real-time facial expression classification is still very long, and on a large part dependent on the computational cost and optimisation of GPUs.

2.3 Face Recognition

2.3.1 Overview

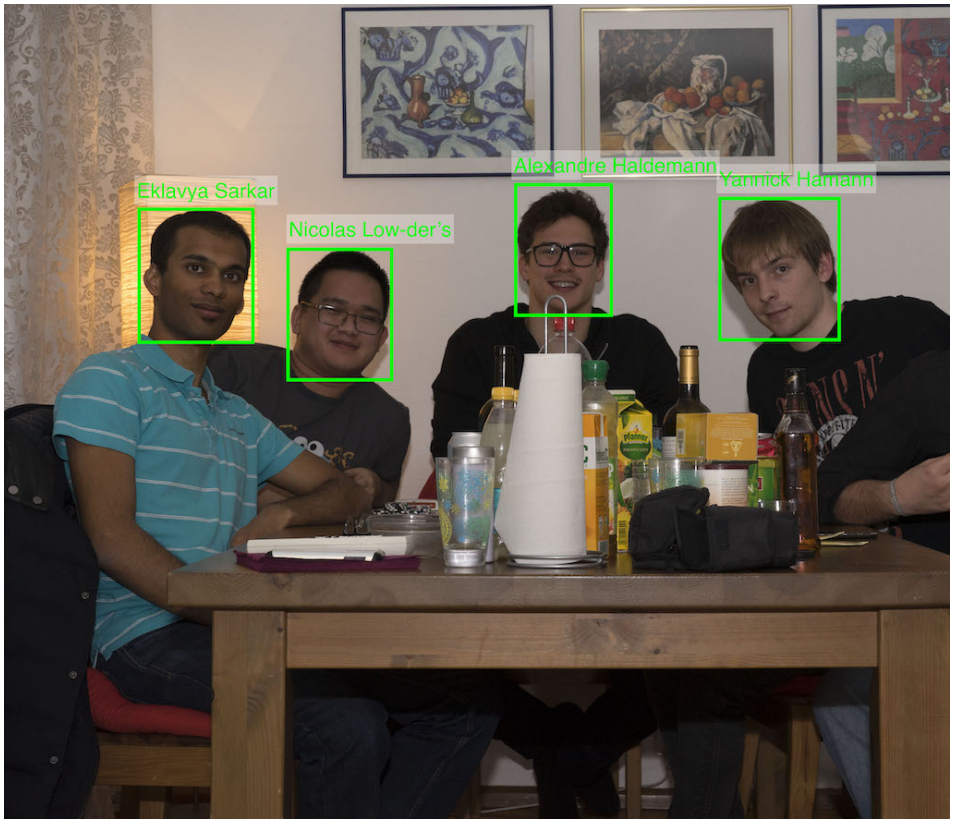

FIGURE 2.8: Sample face recognition on a given image.

Face recognition is the process of successfully identifying a given person's face without knowing their name. This can be both a supervised or unsupervised task, as a model can cluster faces in groups according to similarity, or else categorise each face according to their labels.

Is it a fundamentally different task to the 'anonymous' face detection, and one with the capability of having a much deeper and ground breaking impact on society, as we are currently in the midst of witnessing. It is, however, very much similar to facial expression recognition problem, and has seen practically the same evolution of approaches over time, based on geometric, holistic, and feature-based methods for facial landmark detection. Thus, to avoid redundancy and overlap, this chapter has been placed last, and elements which are not fundamentally unique to face recognition problem have been omitted here.

2.3.2 Challenges

One challenge more so unique to face recognition, than detection or emotion classification, is the process of *ageing*. Indeed, it is still unclear how a model would perform on the recognition of individuals with sample images separated by decades. As recognition is based on facial landmarks, which can distort or change through the years, it is a challenging process to build a model capable of understanding the natural process of ageing and taking it into account.

2.3.3 Applications

Face recognition applications are essentially the same as face detection's, but with more personalised value. It is quickly turning into the preferred biometric method due to its non-invasive process, and could soon be used for most identity authenticate purposes, as we have seen in the latest smartphones such as the iPhone X or the OnePlus 5T. Other applications are not limited to fraud detection, social media tagging, access control, etc.

2.3.4 Traditional Methods

The pipeline for face recognition is almost identical to facial emotion recognition, except the final features are not compared against a set of six predetermined emotions, but simply against other training faces. These methods have generally relied on hand-crafted features, such as edges and texture descriptors, leveraged with machine learning tools such as PCA or SVMs Saez-Trigueros, Meng and Hartnett (2018).

A loss function such as the Euclidian or angular distance could be used for training, and algorithms such as nearest neighbours, metric learning, or threshold comparison could be used for classification during testing. These have traditionally been tested on datasets comprised of famous celebrities' images, as those are far easier to obtain and label, than ones comprised of expressions.

2.3.5 State of the Art

Unsurprisingly, deep learning as impacted face recognition as well, but perhaps more so than for the other two problems. Indeed, some face recognition systems are believed to have in fact now surpassed human performance in some scenarios (Phillips et al., 2018).

Unlike facial emotion problems, huge datasets exist containing 'in-the-wild' images of known people which can be efficiently used in deep learning to improve accuracy of CNN models, and leave behind the traditional holistic or

geometric methods. Taigman et al. (2014a) achieved an accuracy of 97.35% using CNNs, improving on the previous state-of-the-art by an astounding 27%.

There are three main factors that influence the accuracy of a CNN-based method for face recognition: the training data, the architecture of the CNN, and the loss function (Saez-Trigueros, Meng and Hartnett, 2018). While high performance has been derived through softmax's cross-entropy loss function, it has been also been argued that it does not generalise well to subjects not present in the training set.

2.3.6 Conclusion and Future Directions

We have seen here that face recognition is a classification problem not too different from emotion recognition, and techniques which work well with one tend to work well with the other. It is nonetheless not the same problem, and therefore comes with some differences which can be optimised at the hyper-parameter and architectural level. Deep CNNs have obtained high results, and have become the standard, but are also very slow to train and deploy.

In the next chapter 3, we elaborate on the decisions taken on the scope of the project, and review how the collected data was rigorously pre-processed to be prepared for the final implementation stage.

Chapter 3

Data

A primary reason for undertaking this project was to experiment with deep learning techniques on one's own personal data, in order to gain an in-depth understanding of the quality, as well as, the scale of data required for such a modelling task. While there are a number of open-source datasets readily available for such projects, in a real-world scenario data is often collected from scratch, and is considered to be a fundamental part of the data science workflow.

This part of the process is crucial, a critical process, as not only is it important to collect and process the data, but also to understand inspect it as much as possible. We need to understand its distribution and context, and identify patterns and variations.

This chapter first presents a comprehensive overview of the methodology, approach and decisions taken with regard to how the data was collected, stored, processed, and eventually used, before exploring it from a visual and statistical perspective.

3.1 Collection

First of all, it is important to remember that the goal was to develop a *robust* model capable of extracting facial information *in-the-wild*. Therefore, working with a ready-made dataset of cropped faces under optimal lighting conditions specifically for recognition tasks was not considered, even though there are *in-the-wild* datasets are available.

Instead, images were collected from a number of personal sources, such as Google Photos, Dropbox, Facebook, as well as, private photographs dating from ten to fifteen years ago stored on old external hard drives. One of the reasons for using old photographs besides increasing the size of the dataset, was to account for ageing over time, and see how well the model would be able to generalize for subjects with photographic samples spanning more

than a decade.

It is important to note that this data collection process was conducted only to solve the facial recognition and gender classification tasks. Assembling and labelling a large quantity of data for the emotion detection problem would be incredibly challenging, as it is a significantly more complex problem, which is why, the FER2013 dataset was employed instead. It was prepared and pre-processed in the same way as our personal dataset, further explained in-depth in the sections below.

3.2 Bounding Scope

A notable issue when working with the personal dataset was that the sample size distribution was vastly *unbalanced*, even when discarding classes below a certain threshold. Indeed, the two most common classes had around a thousand observations each, whereas the lowest class only a few.

Furthermore, the majority of images in the dataset contained *more than one face*. Hence, even though Google Photos automatically sorted the dataset of images according to each individual's face, additional individual and laborious sorting was required as the model needed labelled and cropped faces for each class during training.

Given the above-mentioned reasons, it was crucial to take some key decisions, early on in the project, with regard to the scope of the model's classification in terms of the final number of classes, as well as, how the data imbalance would be addressed.

Thus, a review of common facial recognition datasets, such as VGGFace2[1] was therefore undertaken to aid decision-making. The review revealed that although the available datasets often contained multi-million number of faces, and thousands of classes, the average number of samples *per class* happened to be roughly around a thousand, which similar to the top classes in our personal dataset.

Moreover, the practical applications of such a model were also rigorously considered, to ensure that none of the time, energy, and effort spent on the project was wasted. For example, it was discovered that if a model was reasonably well-optimised, it could be run on Raspberry Pi through TensorFlow Lite. In which case, it could have multiple useful home-automation

[1] VGGFace2 - A large scale image dataset for face recognition. http://www.robots.ox.ac.uk/~vgg/data/vgg_face2/. (Accessed on 08/10/2019).

usages if run in real-time with the aid of a camera, such as being able to tell who was at the door every time the doorbell rang. Alternatively, the model could potentially be integrated with an Android application for various other purposes and could even delve into the realm of recommendations systems, based on the mood of the subject. The key point for consideration was to focus on collecting data which would help in the development of a model capable of substantially improving the quality of life at home, rather than one proficient at generalising over a large number of individuals.

To this end, it was finally decided to restrict the scope of the project only to my immediate family - comprising four individuals - myself, my sister, mother and father, and who also happened to be the ones with the highest number of samples. Moreover, issues around data consent and privacy were easily addressed, as my family fully supported and consented to participating in the project. Nonetheless, starting a deep learning project with a relatively small dataset was a daunting prospect, with a lot of unknowns, and requiring further engineering. The next sections describe, in depth, the challenges encountered and the solutions formulated.

3.3 Preparation

Once the data was collected from the various sources, it needed to be prepared in a regulated format for all models. The first step was to meticulously sort all the images by classes, even though most images contained multiple faces, inducing multiple crossovers between classes. The next step was to crop and save only the faces in all the images, again sorted by class. To this end, a small Python script, `prepare_cropped_faces.py`, given in section D.10, was written, which would look at all the images in a given directory, and output images of cropped faces on the given output directory.

FIGURE 3.1: Sample image and corresponding cropped faces. `OpenCV` can occasionally miss real faces due to partial occlusion (false negatives).

A number of off-the-shelf libraries were suitable for this task, including ones which used convolutional neural networks[2], however `OpenCV` was eventually used for the task because of high-speed performance , as well as, ease of access (uncommon packages tend not to be pre-installed in environments).

Once the 28x28 cropped faces were available in a new folder, these again had to be manually sorted, to remove the numerous false positives detected by `OpenCV`, as well as, faces which did not belong to the specific class in question. Once the process was repeated for all four classes, the images were double-checked multiple times as part of the 'data cleaning' process to ensure there was an absolute 0% error rate in the class labels.

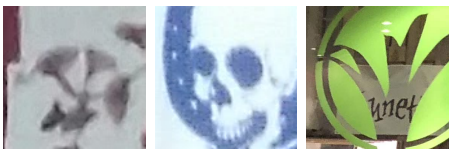

FIGURE 3.2: False positives samples marked as faces by `OpenCV`.

The labeling of each class was done in a simplified manner while importing the data into the model. A list of ground truth labels was simply created in Python according to the length of each class size, e.g. 1000 labels of class 0 for the first class, 800 labels of class 1 for the second class, etc.

3.4 Storage

Once the final, clean data was available in directories, one could simply import them into Python using `OpenCV`'s `cv2.imread()` method. However, the lack of vectorisation slowed down the process, and it would take up to 40 minutes in total to import all classes - a significantly inefficient process for a relatively small dataset.

Although `OpenCV` proposed no direct method to store colour images, due to their 3-dimensional nature, an elegant solution was to simply reshape each image's pixels, a task made easily possible with `NumPy`. Thus, another script, `convert_images_to_data.py`, shown in section D.11, was written to reshape all the images in a given input directory, and output them as a `.csv` file. Each row of this file would contain the data of a single 28x28x3 cropped face image, reshaped to a single `(2352,)` line.

This method allowed the dataset of images to be neatly loaded in a `NumPy` array in less than a minute, representing a 40x increase in loading speed.

[2]Multi-Task Convolutional Neural Network https://github.com/jbrownlee/mtcnn. (Accessed on 08/10/2019).

3.5 Pre-processing

The following steps were taken within the models at each execution, after the data was imported from the `.csv` files for each class.

3.5.1 Conversion to RGB

Firstly, the entire data of each class was loaded into the memory though Pandas' `read_csv` method. Since `OpenCV` was employed for face detection, and read images in `BGR`, as opposed to `RGB`, the data was converted to the latter format with `cv2.cvtColor(image,cv2.COLOR_BGR2RGB)`. No computation is needed for this conversion, as the data is simply re-arranged.

FIGURE 3.3: Sample image in BGR and RGB format.

3.5.2 Histogram Equalisation

The next step was to perform histogram equalisation, a process which increases the global contrast of an image by levelling its histogram. It can produce results which may look odd to the human eye, but are useful from a numerical point of view, as it is easier to differentiate between features when they are more strongly accentuated. A histogram of an image can be calculated with `cv2.calcHist(images,channels)`, and then accordingly adjusted with `cv2.equalizeHist(image)`.

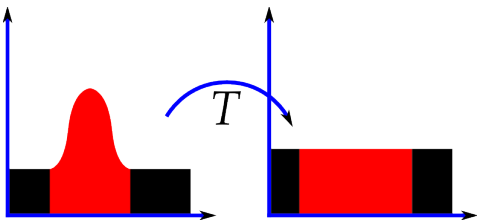

FIGURE 3.4: Histogram equalisation concept through a transformation T. Taken from Wikipedia (2019)

The mathematics behind histogram equalisation is relatively easy to grasp, and can be explained through the equations below. We assume $H(p)$ is the image's current histogram, ranging over $[p_0, p_k]$, which constitute the intensity of a channel. $H(p_i)$ thus represents the number of pixels with level

p_i. Similarly, $T(q)$ represents the transformed equalised histogram, ranging over $[q_0, q_k]$. There are n number of pixels, ranging from q_0 to q, which in the case of an un-normalised image would be 0 and 255, respectively. The equalised histogram is the same as the original one, but simply with its values evenly distributed. Consequently, the total number of pixels remains the same, as summarised by Equation 3.1.

$$\sum_{i_0}^{k} T(q_i) = \sum_{i_0}^{k} H(q_i) \tag{3.1}$$

Furthermore, as T follows a uniform distribution, we can derive its cumulative distribution as shown in Equation 3.2.

$$T(q) = \frac{n}{q_k - q_0} \tag{3.2}$$

We can modify Equation 3.1 to be represented in a continuous space, in order to derive the monotonic transformation function τ which equalises the histogram. As we can see, the derived calculation is not computationally intensive. The original p values are simply 'shifted' to the new q values.

$$\begin{aligned}
&\int_{q_0}^{q} T(s)ds = \int_{q_0}^{q} H(s)ds \\
&\Leftrightarrow \int_{q_0}^{q} \frac{n}{q_k - q_0} ds = \int_{q_0}^{q} H(s) \\
&\Leftrightarrow \frac{n}{q_k - q_0} \int_{q_0}^{q} ds = \int_{q_0}^{q} H(s) \\
&\Leftrightarrow \frac{n(q - q_0)}{q_k - q_0} = \int_{q_0}^{q} H(s) \\
&\Leftrightarrow q = \frac{q_k - q_0}{n} \int_{q_0}^{q} H(s) + q_0 \\
&\Leftrightarrow q = \tau = \frac{q_k - q_0}{n} \int_{q_0}^{q} H(s) + q_0 \\
&\Leftrightarrow q = \tau = \left(\frac{q_k - q_0}{n} \sum_{i=p_0}^{p} H(i) \right) + q_0
\end{aligned} \tag{3.3}$$

It is to be noted that equalisation with `OpenCV` can only be done on a single channel at a time, as it is usually used on grey-scale images. However, applying it per channel allows us to equalise an image in different ways. The `cv2.split()` method, as shown in Listing 3.1, separates an image into the desired type of channels, e.g. RGB, HSV, and YCrCb. This allows us to select and operate precisely on the required channel, resulting in different final results, as demonstrated in Figure 3.5.

```
1 def equalise_image(self, image, eq_type='HSV'):
```

```
    if eq_type == 'HSV':
        H, S, V = cv2.split(cv2.cvtColor(image, cv2.COLOR_RGB2HSV))
        eq_V = cv2.equalizeHist(V)
        return cv2.cvtColor(cv2.merge([H, S, eq_V]), cv2.
    COLOR_HSV2RGB)
    elif eq_type == 'YCrCb':
        Y, Cr, Cb = cv2.split(cv2.cvtColor(image, cv2.
    COLOR_RGB2YCrCb))
        eq_Y = cv2.equalizeHist(Y)
        return cv2.cvtColor(cv2.merge([eq_Y, Cr, Cb]), cv2.
    COLOR_YCrCb2RGB)
```

Listing 3.1: Different ways of equalising histograms for global contrast correction.

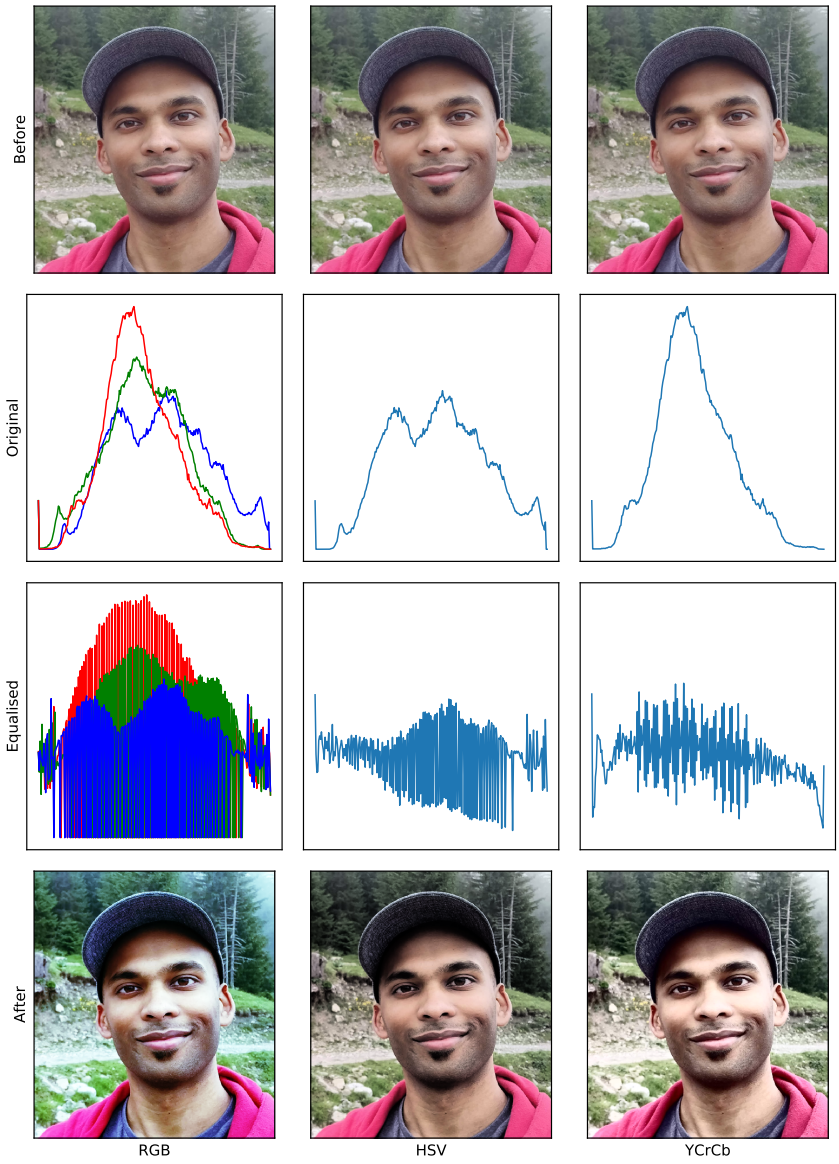

Figure 3.5: An image modified in three different ways, using global contrast equalisation, with their respective histogram distributions.

There are a other more sophisticated histogram equalisation techniques, such as Adaptive Histogram Equalization (`AHE`) and Contrast Limited Adaptive Histogram Equalization (`CLAHE`), which employ adaptive methods, as opposed to ones which work on the global contrast. `CLAHE` works similar to 'tile-coding', i.e. an image is divided into a given number of tiles, each of which is then independently equalised. That way histograms are confined to only a small portion of the image, unless there is a certain amount of noise. In which case, *contrast limiting* is applied, which clips off contrast values which are above a certain threshold, and distributes them over other bins.[3]

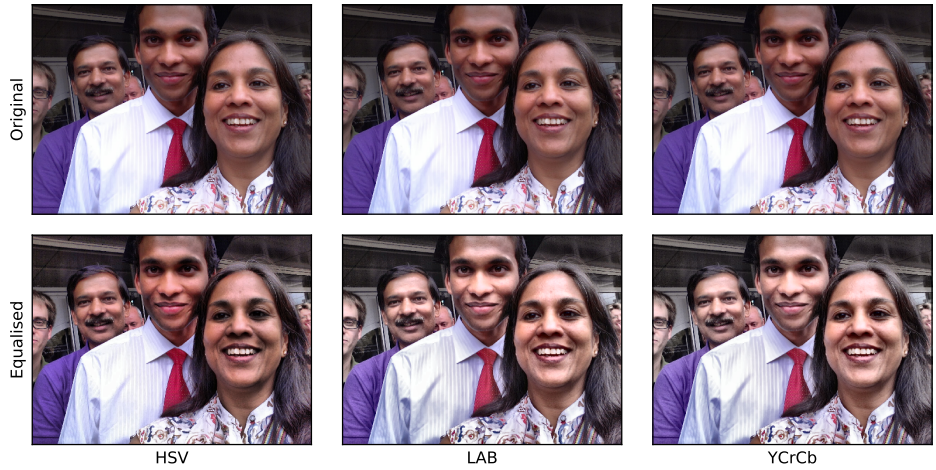

FIGURE 3.6: Channel image equalisations using `CLAHE`.

For our model, global contrast adjustment was chosen over more sophisticated methods, as this needed to be performed on an already cropped region of a photo. From the available methods, `HSV` equalisation channels were preferred over `RGB` or `YCrCb` ones, as the former yielded the best visual results.

3.5.3 Normalising Data

The image values have by now been modified multiple times, but are still in the $[0, 255]$ range. The next step was to normalise all our data to, either between $[0, 1]$, or $[-1, 1]$. It is highly recommended to keep all the input values small, to avoid having large gradient updates which can prevent a model from converging. Furthermore, all the input features should be in the same scale, so that the model is not biased towards a feature simply because it has larger absolute values. This also ensures faster convergence in training during the gradient descents.

[3] OpenCV: Histograms - 2: Histogram Equalization https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html (Accessed on 08/11/2019).

Although this task sounds trivial, in truth it can be done in several ways, which surprisingly, produce substantially differing results. The first decision to make was whether to normalise over the entire image, or over each feature independently - in our case for example, over the R, G, B channels.

Another choice, was to decide whether to normalise the data between $[0, 1]$, rescale (between $[-1, 1]$, or else standardise to have a mean of $\mu = 0$ and standard deviation of $\sigma = 1$. As all these choices are mutually exclusive, only one of them can be selected. In addition, if selected, the standardisation method could be executed on each image independently (fixed-image normalisation), or over the entire training dataset, with the test dataset later adjusted with the same parameters μ and σ (group normalisation).

Eventually, the fixed-image and feature-independent standardisation method was preferred, as it first centred the data of each channel around the mean for each image, before bounding it with a maximum deviation of 1.

```
def standardise_image(self, image):
  '''Standardise an image per channel'''
  means = image.mean(axis=(0,1))
  stds = image.std(axis=(0,1))
  return (image - means) / stds
```

Listing 3.2: Used method for standardising data.

3.6 Augmentation

Despite all the measures taken so far to 'improve' our data, it was still far from sufficient for training multiple *conventional* convolutional networks. The two principal reasons for this were the low quantity of data, and the data imbalance.

Nonetheless, it is important to bear in mind, that the quantity of required data for the model to produce competent results is also *relative to the complexity of the model*, i.e. the size and depth of the network that it would train on. This is due to the very nature of convolutional neural networks and their ability to learn local and translation invariant features, which allows them to be highly data-efficient on computer vision tasks.

Therefore, in order to fully ensure no issues arose, and to build a robust foundation for all models, we approached the problem from both sides. On the one hand, we kept our model relatively simple and shallow for the facial recognition and gender classification tasks (explained in-depth in chapter 4),

and on the other hand, we used data augmentation to increase the total number of samples, in order to more accurately equalise the data imbalance between classes.

Data augmentation is a technique which performs many simple affine transformations on existing images to produce similar, but *augmented* ones, which act like new training data for the model. Furthermore, it also helps to counter over-fitting in models which could learn to simply memorise the training data and lose the ability to generalise on unseen images.

This can be implemented in several ways, some easier than others. For example, Keras, a deep learning library, contains the `ImageDataGenerator()` method which allows easily configurable data augmentation in real-time, i.e. as the model is training. Alternatively, there are other modern Python libraries, not specifically linked to Keras or TensorFlow, such as `ImAug`, `Augmentor`, and `Albumentations`, developed by Jung (2018), Bloice, Roth and Holzinger (2019), and Buslaev et al. (2018) respectively.

To fully understand and control the process, as well be able to closely fine-tune the parameters, manual data augmentation, specifically `Albumentations`, was preferred over the more automated processes such as with Keras.

It was hard to find a precise rule-of-thumb for data augmentation, but we decided to augment each class by a certain factor, each of which would be 70% *weak* augmentations and 30% *strong* ones. Both of these have a certain number of total possible applicable transformations, which can be randomly applied based on defined probabilities. Thus, each augmentation is a different, random, and likely unique overall transformation, defined by a number of small affine transformations. The distinctive feature between weak and strong augmentation is that the latter contains 'grid distortions', a transformation which bends an image's grid-lines in a non-linear way, and thus distorts the image more so than any other method.

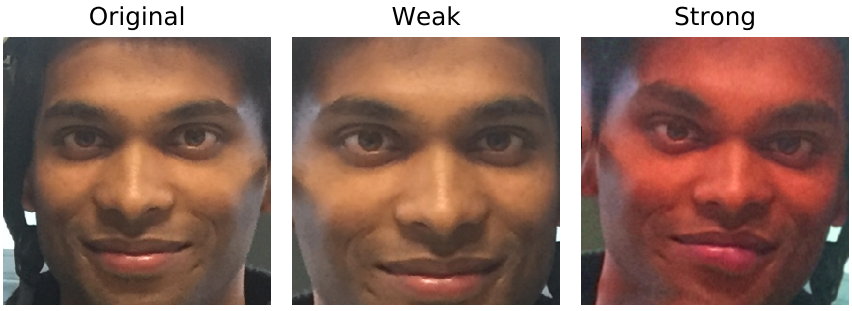

FIGURE 3.7: Sample image compared with possible augmentations.

Note that neither of the two augmentation types apply any right angle rotations, transposes, or vertical flips, as convolutional neural networks are not invariant to rotations despite being invariant to translations. Indeed, the weak and strong augmentations have a rotation limit of 10 and 45 degrees respectively. The final code used for creating weak augmentations is shown in Listing 3.6. Other common transformations include colour changes, motion blurs, Gaussian noises, optical distortions, and interestingly `CLAHE`.

```
def weak_aug(self, p=0.5):
    '''Create a weakly augmented image framework'''
    return A.Compose([
            A.HorizontalFlip(),
            A.OneOf([A.IAAAdditiveGaussianNoise(), A.GaussNoise
    (),], 
            p=0.2),
            A.OneOf([A.MotionBlur(p=0.2), A.MedianBlur(
    blur_limit=3,
            p=0.1),
            A.Blur(blur_limit=3, p=0.1),], p=0.2),
            A.ShiftScaleRotate(shift_limit=0.0625, scale_limit
    =0.2,
            rotate_limit=10, p=0.2),
            A.OpticalDistortion(p=0.2),
            A.OneOf([A.CLAHE(clip_limit=2), A.IAASharpen(),
            A.IAAEmboss(),], p=0.3),
        ], p=p)
```

This task could have also been pre-processed and stored in the dataset in order to speed up the process of modifying data, but as augmentation is most often performed *during* training, it was preferred to keep it inside the model, in order to keep the augmentations random and different each time.

3.7 Exploration

Once our dataset is processed, augmented, and finally static, we can explore the data from a visual perspective, as well as, by looking at the actual counts and relative frequencies. Figure 3.8 and 3.9 show the different classes of the datasets as bar and pie charts.

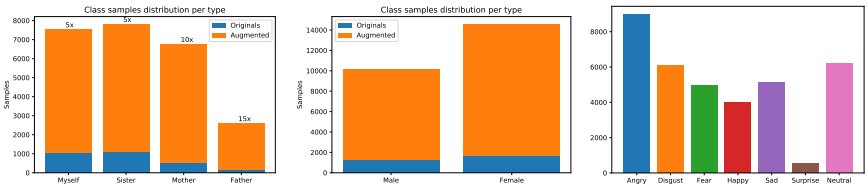

FIGURE 3.8: Datasets class distributions viewed as bar charts.

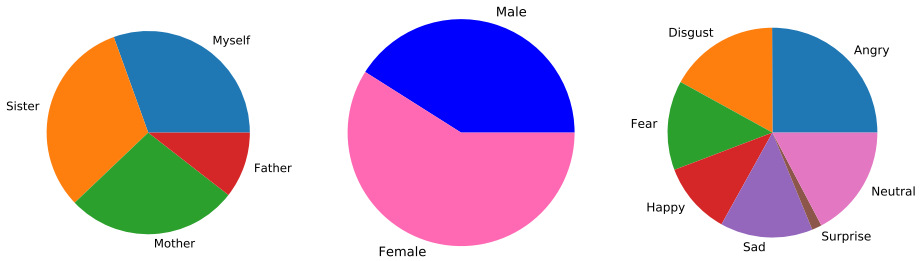

FIGURE 3.9: Datasets class distributions viewed as pie charts.

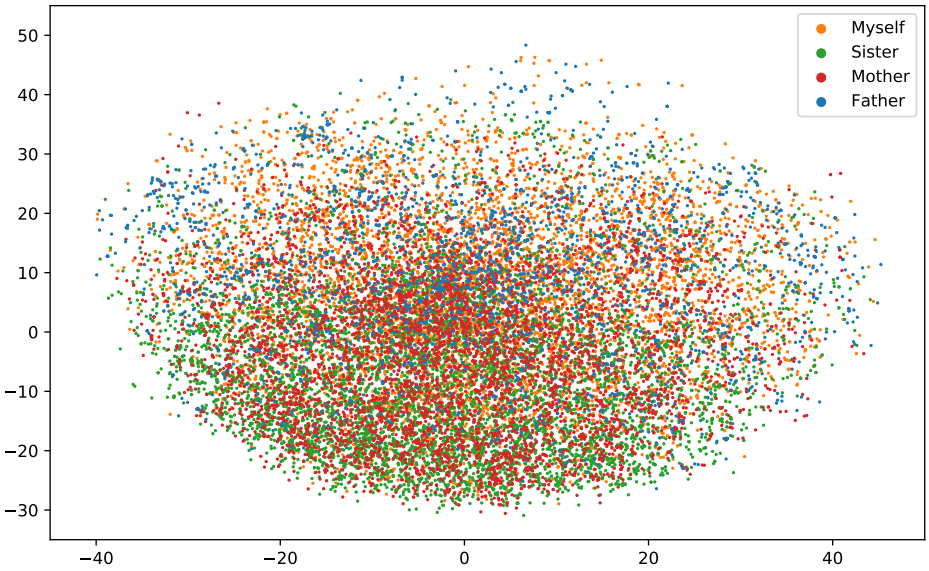

FIGURE 3.10: PCA plot of the dataset sorted per classes.

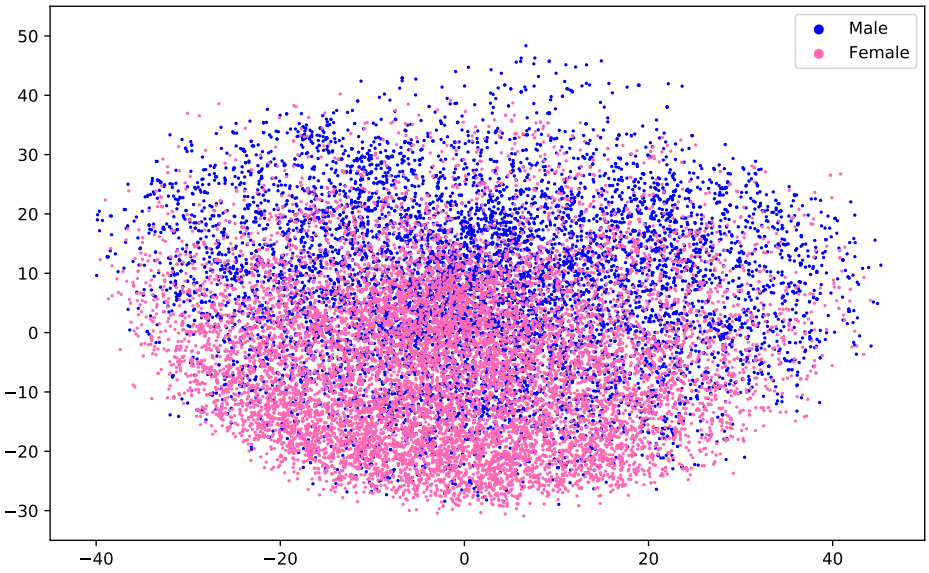

FIGURE 3.11: PCA plot of the dataset sorted per gender.

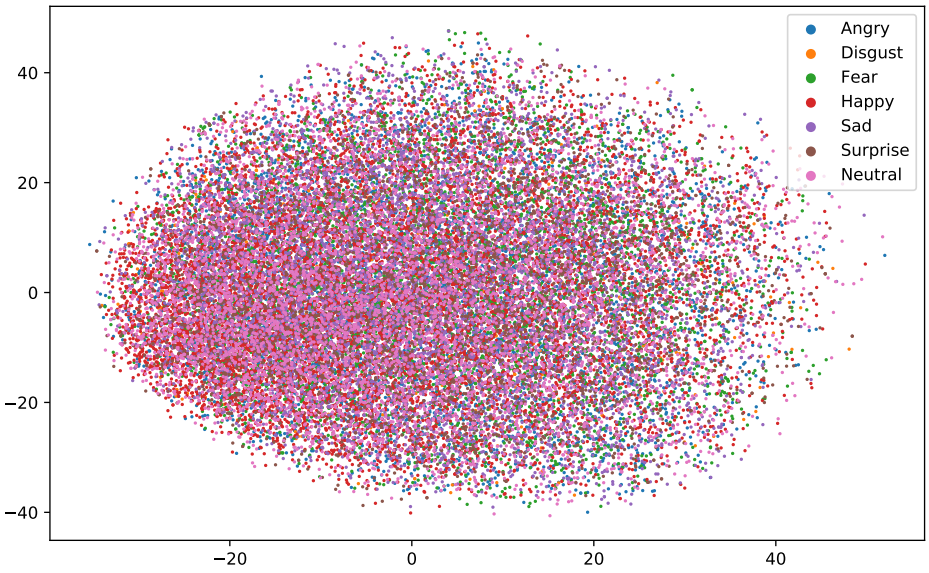

FIGURE 3.12: PCA plot of FER2013 dataset sorted per emotions.

We can also visualise the data by conducting Principal Component Analysis on the dataset, and colour coding it according to the classes, as shown on Figure 3.10, 3.11, and 3.12. This reduces the dimensionality of the data from a four-dimensional space (number of images x 28 x 28 x 3) to a two-dimensional one, allowing us to plot and visualise the whole data space.

Finally, we report the actual counts of each class in their respective databases, as given below in Table 3.1, 3.2, and 3.3, alongside their respective relative frequencies.

TABLE 3.1: Count and relative frequencies of dataset by classes.

Class	Original	Augmented	Total	Frequency
Myself	1078	6468	7546	0.305
Sister	1117	6702	7819	0.316
Mother	564	6204	6768	0.274
Father	153	2448	2601	0.105
Total	2912	21822	24734	1.0

TABLE 3.2: Count and relative frequencies of dataset by gender.

Class	Original	Augmented	Total	Frequency
Male	1231	8916	10147	0.41
Female	1681	12906	14587	0.59
Total	2912	21822	24734	1.0

TABLE 3.3: Count and relative frequencies of FER2013 by emotions.

Class	Total	Frequency
Angry	8988	0.251
Disgust	6076	0.169
Fear	4945	0.138
Happy	4001	0.112
Sad	5121	0.143
Surprise	547	0.015
Neutral	6197	0.173
Total	35875	1.0

This gives us a concrete idea of the data we shall be working with, sending it to our models to train with, and finally predict on unseen data. In the next chapter 4, we shall look into designing the models themselves.

Chapter 4

Design and Engineering

Thus far, we have directly worked on the data - processing and augmenting it till a satisfactory level. However, when designing the actual models themselves and their surrounding 'systems', it is important to build them with strong software engineering principles from the very outset. As neural networks are not an 'off-the-shelf' technology and can often fail silently in a number of ways, they require considerable thought during the design and engineering phase. The sections below give an overview of all the key design principles and guidelines used in building the system and models, an intensively demanding and time-consuming task. The reasoning behind each decision also gives an insight into the iterative 'design-and-implement' workflow, as well as, our approach towards the various issues which needed to be continuously resolved to achieve the final product.

4.1 Workstation

The first and foremost step before actually even working on the models was to establish a stable workstation conducive to a productive workflow environment. To this end, two different alternatives were possible, as explained in the following paragraphs.

4.1.1 Local

The initial Python scripts for data pre-processing were written on a simple text editor, namely Sublime Text, which was the easiest and light-weight method. However, it quickly became limited for any tasks which demanded visualisations and quick turnovers. To this end, Jupyter notebooks were preferred, as the modular and individual cells were more practical for our iterative development methodology and rapid debugging.

However, the actual training process on a local laptop was far from ideal, as the hardware wasn't optimised for deep learning in any way, nor did it contain any graphical processing units to speed up the process. Training for

50 epochs could easily take up to an hour, only to sometimes not converge because of a silent bug.

4.1.2 Kaggle

Kaggle kernels were therefore chosen for the faster processing speed and state-of-the-art hardware. As of August 2019, the specifications were as given in Table 4.1.

Table 4.1: Hardware specifications of Kaggle kernels.

Hardware	Value
NVIDIA-SMI	418.67
Driver Version	418.67
CUDA Version	10.1
GPU	Tesla P100-PCIE-16GB
GPU Name	Persistence-M
CPU Cores	4
CPU RAM	16 GB
GPU CPU Cores	2
GPU RAM	13 GB

While working with Kaggle kernels allowed for faster model convergence, it also slowed down productivity on some other aspects. For instance, the entire notebook had to be 'committed' (run) from top-down, in order to produce any output files, such as graphs, weights, checkpoints, etc. which was particularly exhausting for simple situations, for example when only a single plot was required.

Furthermore, in order to import a custom class, i.e. one written by ourselves, and which included useful methods common to all models, one had to import it as a 'utility script' at the initialisation of a Kaggle kernel. If a change was to be made in the utility script, then it had to be edited on Kaggle, saved, and then required the model notebook to be restarted. While this is not too different from a local workflow, the multiple reloads and commits often made debugging and adding new features particularly tedious and slow, even as it speeded up the training and also improved the speed of general executions.

Nonetheless, the utility scripts and multiple models were all eventually developed using the Kaggle kernels as workstations. Moreover, the commit history of each model gave a clear view of the incremental evolution, which also provided some insight into the software development methodology.

4.2 Modularity

The first and foremost design principle was to keep everything modular. This was particularly relevant in our context, as we had three concrete tasks to solve, each of which could have several models. In order to reduce redundancy and keep debugging at a manageable scale, class templates were designed for the most common methods.

This idea, like most software development decisions, was a package deal which came with its own set of pros and cons. On the one hand, the code used on different notebooks was identical yet non-repetitive, and much more manageable. But on the other hand, the process of building classes separately, ensuring their flexibility, and debugging and testing them on other notebooks was a much more iterative and slower process than simply writing code directly on the current notebook. It was also important to maintain consistency over *all* the methods, in order to be able to compare results across models. Similarly, we had to ensure the output items, such as graphs, weights, and checkpoints were all saved with a similar formatting style, such as the DPI value, grid structure, colours, etc.

Eventually, the following three external classes were developed in order to aid the tasks at hand:

- `myutilitymethods.py`, given on section D.3.
- `mycnn.py`, given on section D.5.
- `mydeepcnn.py`, given on section D.6.

The key to developing `myutilitymethods.py` was understanding the fact that all the three tasks - face recognition, gender classification, and emotion detection - were all *bounded*, i.e. had a pre-defined number of class variables. For instance, face recognition had a total of 4 distinct classes, gender classification had 2, and emotion detection 7. Each of these could simply be defined by three Python variables, as shown in Listing 4.1.

```
class_to_num = {
    'Myself': 0,
    'Sister': 1,
    'Mother': 2,
    'Father': 3
}
num_to_class = ['Myself', 'Sister', 'Mother', 'Father']
NUM_CLASSES = 4
```

Listing 4.1: The core variables for the facial recognition model.

These are effectively all the essential variables required by a model to be able to use the common methods. An additional `class_colors` list variable could be used if we want each class to have a pre-defined colour, but it is not indispensable. The common methods class `myutilitymethods` can thus be loaded in the model notebook with the core variables as shown in Listing 4.2.

```
from myutilitymethods import MyMethods
mm = MyMethods(NUM_CLASSES, num_to_class, class_to_num)
```

LISTING 4.2: Importing and instantiating `myutilitymethods`

We now have access to all the methods defined in the `myutilitymethods.py` file, imported as a `MyMethods` class, and instantiated as `mm`. Once an understanding of the core variables was defined, and the flow of data clearly established, it was a straightforward, albeit tedious, process of simply adding new methods to the `MyMethods` class, which would be callable across any model. As such, the first few methods to be written were the data preprocessing ones, such as `convertToRGB()`, `equalise_image()`, `augment_images()`, `standardise_images()`, as given earlier in chapter 3.

It is to be noted, that all the methods which required external libraries, such as `sklearn` or `TensorFlow` had to be called inside the `myutilitymethods.py` file itself, and not the model notebook, to be functional. This process was also the one where the utility and importance of clear documentation fully came into play. Which is why, each method includes a *docstring* and inline comments to explain the flow of data. The full list of methods in the class, and their respective docstrings, can thus easily be seen with the `dir(MyMethods)` or `help(MyMethods)` commands.

4.3 Reproducibility

The second main design principle was to ensure reproducibility. Indeed, each new component added to a model had to not only be modular, but also work in the anticipated way, with no unexpected or random changes. This meant engineering systems which could take in pre-defined seeds as parameters and use them for any actions which required random values. For instance, the data is not only randomly shuffled at the very start after initially loading it from the `.csv` file, but also at the start of each epoch during the training phase, as demonstrated in Listing 4.3.

```
from sklearn.utils import shuffle

#Shuffle order for both subsets inside a class
```

```
4 x_in, y_in = shuffle(x_in, y_in, random_state=self.seed)
5 self.x_val, self.y_val = shuffle(self.x_val, self.y_val,
      random_state=self.seed)
```

Listing 4.3: Mixing different data batches with the same seed.

Having a pre-defined seed helped ensure a result that was never a one-off and was always replicable. This was particularly relevant as we wanted to clearly show the evolution and growth of the system over time, which therefore required previous results to still be obtainable. The `mycnn.py` and `mydeepcnn.py` classes were both designed and developed with these ideas in mind, which helped ensure that as new modular features were being added to the systems, there was always an option of not using them, and obtaining the previous results again.

4.4 Flexibility

Another important design principle was flexibility. Specifically, this is in the context of the quantity of data, which can theoretically vary throughout the development of the project. Indeed, it was in our interest to keep collecting data, even if we have completed data pre-processing, as this is arguably the most certain way of improving the quality of a model. Additionally, one could have decided to increase number of classes later, as and if we wanted to add individual subjects in order to expand the scope of our project.

Thus, it was important to develop a system which could receive a flexible amount of inputs. This might sound straightforward, but when training or testing the model in *batches*, having several samples which cannot be directly divided by the batch size can be tricky to handle. Furthermore, the actual need of batches can vary depending on the quantity of input samples, as the optimal batch-size depends on the workstation's hardware specifications. In many cases, we discovered that it was actually more optimal to have batches sizes as a *constant* value (as opposed to a flexible one), when dealing with a number of samples which were below a certain threshold, as these would directly fit into a GPU or CPU's cache.

One had the choice of hard-coding the batch sizes to a set constant, inputting it in the model class as a parameter, or else automating the process by having the size directly vary as a function of the total number of input samples. Eventually, the option of having it as a class parameter variable was chosen, as many sources cited limiting the training batch-size to around 30, but recommended tuning it according to the size of the inputs and other model parameters. Additionally, batches were simply discarded

during testing for prioritizing speed, as the batches were only really useful for optimising the gradient descents during the training.

4.5 Validation

As with any software engineering task, one has to ensure that it will not crash in different test case scenarios, and is robust enough to competently handle unexpected inputs.

For neural networks, it is very important to always know what is being fed to the network at the very `start`. For example, our CNN model takes in the labels as a `(num_samples,num_classes)` array of one-hot encoded values. If the standard, non-encoded, one-dimensional list of labels is provided instead, the network could potentially still execute without giving a direct error; and, depending on the nature of the methods employed, make the ensuing debugging process of 'silent failures' all the more harder.

Therefore, the most efficient method for catching such bugs is in fact to use assert statements which verify the expected shape of the input data before it is fed to the model. In addition, we can also use assert statements to ensure that the previous data pre-processing successfully modifies the data as expected, such as converting the mean μ to 0, and standard deviation σ to 1, as shown in Listing 4.4. This ensures that obvious errors are caught and dealt with, well before the system starts training.

```
# One-hot encode
y_train = mm.one_hot_encode(y_train)
y_test = mm.one_hot_encode(y_test)

# Assert shapes
assert(y_train.shape[1]==NUM_CLASSES)
assert(y_test.shape[1]==NUM_CLASSES)

# Assert std devs
assert(np.round(x_train[0].std(), 3) == 1)
assert(np.round(x_test[0].std(),  3) == 1)

# Assert means
assert(np.round(x_train[0].mean(), 3) == 0)
assert(np.round(x_test[0].mean(),  3) == 0)
```

Listing 4.4: Code to check various shapes and values.

The model can still fail silently if there are fundamental issues with its engineering design. However, we can nonetheless look into resolving some

of these issues, which are related to other class parameters and hyper-parameters. The `model_summary()` method gives the model's architecture by printing out the shapes of each of its layers, as shown in Listing 4.5.

```
first_layer   : conv2D  - (?, 25, 25, 32)
second_layer  : conv2D  - (?, 11, 11, 64)
third_layer   : conv2D  - (?, 4, 4, 128)
fourth_layer  : conv2D  - (?, 1, 1, 256)
flattened.    : Flatten - (?, 256)
logits        : Dense   - (?, 7)
preds         : Softmax - (?, 7)
```

Listing 4.5: Values returned by `model_summary()`.

Similarly, the `model_variables()` method prints out the list of input parameters, defined by the user, as given in Listing 4.6.

```
x_train              : (34081, 28, 28, 3)
y_train              : (34081, 7)
x_test               : (1794, 28, 28, 3)
y_test               : (1794, 7)
output_dir           : ./FER_logdir/
lr                   : 5e-05
nb_epochs            : 50
batch_size_train     : 30
seed                 : 0
nb_classes           : 7
nb_images            : 34081
nb_train_iterations  : 1136
im                   : Tensor("Placeholder:0", shape=(?,28,28,3))
labels               : Tensor("Placeholder_1:0", shape=(?,7))
```

Listing 4.6: Values returned by `model_variables()`.

Both these methods were indispensable tools for debugging errors or testing different scenarios when building the `mycnn.py` and `mydeepcnn.py` models.

4.6 Incremental design

Despite the prevention measures put in place, several issues still remain, which can cause neural networks to fail silently. Some of the common ones which are listed below.

- Off-by-one error.
- Inputs and labels not shuffled with the same seed.
- Provided inputs without pre-processing or with incorrect shapes to model.

- Mismanaged parameters such as regularisation, learning rate, decay rate, etc.
- Didn't account for augmented data when making labels for input data.
- Mixed up different `tf` sessions, graphs and checkpoints.
- Passed softmax outputs to a function which expected raw logits.

The best methodology to avoid such issues is to never let them happen in the first place. Indeed, our last principal design policy and the fundamental blueprint for building our neural networks was to *keep things simple*. This meant: first ensuring that the initial results were in line with what was expected, and only then hypothesise on what an additional feature might do. After we implemented it, we again ensured that the new results were as expected, and so on. Adding a lot of initial *unverified complexity* such as drop-out layers, batch normalisation, back-tracking, etc. would have needlessly slowed down any potential debugging process, and was thus carefully avoided until a simpler baseline method could first be established.

As we have shown in this chapter, the design and engineering part of this project included a lot of careful thought and experimentation, which would often validate or discredit new features, and thus influenced the final software. This was a fundamental part of the project, and was given as much time, importance, and appreciation as the more scientific aspects relevant to data science, and is further elaborated in the next chapter 5: Implementation.

Chapter 5

Implementation

This chapter provides a comprehensive and more granular review of all the implemented items which especially played a critical role in the development of the two network classes `mycnn.py` and `mydeepcnn.py`. This not only includes elements such as the parameters, hyper-parameters, validation methodologies, architectures, and metrics, but also the experimentation and reasoning behind the decisions to use them. In this chapter, we consolidate all the knowledge and logic behind the development of the various models using the two network classes, in order to finally solve our three tasks.

5.1 Networks Architectures

5.1.1 My CNN

The fundamental idea behind the `MyCNN` network design was to keep it as simple as possible. Indeed, in consideration of the low amount of available data (relative to known open-source datasets which have millions of samples), the network was chosen to be shallow and contain the least amount of complexity as possible. We in fact wanted to determine how well a model of only a few layers would work on a dataset of around 22000 samples. As the total number of classes of two of the main tasks was small - only four for facial recognition and two for gender classification - we hypothesised that the complexity described in the paragraphs below would be sufficient for these two tasks.

Bearing this in mind, we therefore decided to only have 4 convolutional layers with rectified linear unit (ReLU) activations acting on our 28x28x3 `RGB` input image, which would be passed on to a fully connected layer to obtain the raw logits. Finally, the outputs were acquired in the last layer through the means of a softmax activation function. The design principles described in chapter 4 also allows a user to alternate between a softmax and a sigmoid activation function for the final layer if necessary. This flag switches the optimiser's function between a softmax or sigmoid cross entropy loss.

The details of this specific architecture can be seen in Table 5.1, and visualised in Figure 5.1. It is to be noted that the latter shows the final logits and outputs if applied to the *emotion classification* task, as the final number of classes shown is 7. For the facial recognition or gender classification tasks, the final layer's size would be 4 or 2 respectively.

TABLE 5.1: `MyCNN` architecture table.

Layer	Input	Filter	Stride	# Filters	Activation
Input	28x28x3				
Conv1	5x25x32	4x4	2	32	ReLU
Conv2	1x11x64	4x4	2	64	ReLU
Conv3	4x4x128	4x4	2	128	ReLU
Conv4	1x1x256	4x4	2	256	ReLU
FC	256				Linear
Logits	`num_classes`				Linear
Output	`num_classes`				Softmax

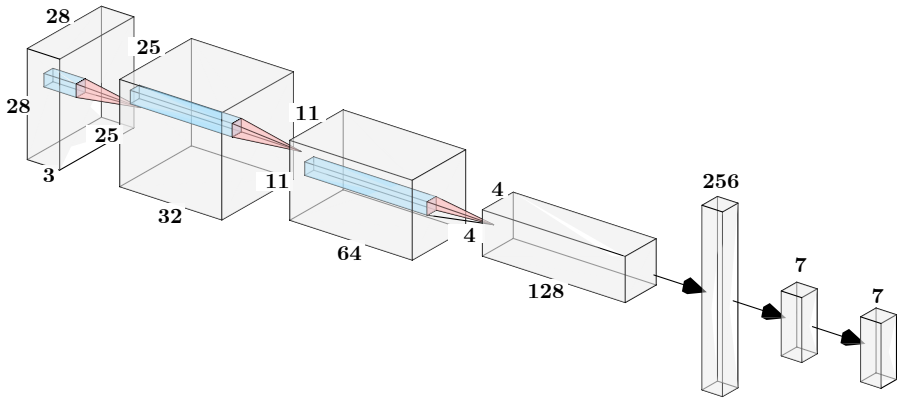

FIGURE 5.1: Diagram of `MyCNN` architecture.

The idea of using a shallow network was based on the convolutional neural networks used to model on the MNIST dataset, which also contained 28x28 images, albeit in greyscale instead of RGB. The same architecture has been previously shown by Megagenta and Sarkar (2019) to get 98.9% accuracy on MNIST classification. The idea was to experiment with a similar architecture and see if good results could also be reproduced on our personal dataset, by means of hyper-parameters tuning.

The `MyCNN` network, although a simple one, was built using TensorFlow, which has a relatively steep learning curve, especially because of the multiple

and separate writer, saver, and graphs variables. The network was implemented with three set-up functions - `set_up_saver()`, `compute_loss()`, and `optimizer()`, which initialised the variables mentioned earlier; and three core functions - `create_model()`, `train()`, and `test()`, which worked on the input data variables such as `x_train`, `y_train`, `x_test`, and `y_test`.

5.1.2 My Deep CNN

In the event where the shallow network `MyCNN` was unable to generalise well to a complex task, such as emotion classification, a second, deeper network was also prepared, which made use of much more advanced deep learning techniques, such max-pooling, dropouts, and decaying learning rates. As represented in Figure 5.2 (also in Figure F.1), and shown in Table 5.2, a total of 8 convolutional layers were used, as well as, 4 max-pool and 3 dropout ones. The weights were then flattened, before an additional 3 alternating dense and dropout layers were employed. Each convolutional layer had the parameter '`same`' as padding, which ensured that the output of the layer in question had the same length as the input. In this model, both the convolutional *and* dense layers were activated with ReLU functions. Finally, the output was calculated using a softmax activation function.

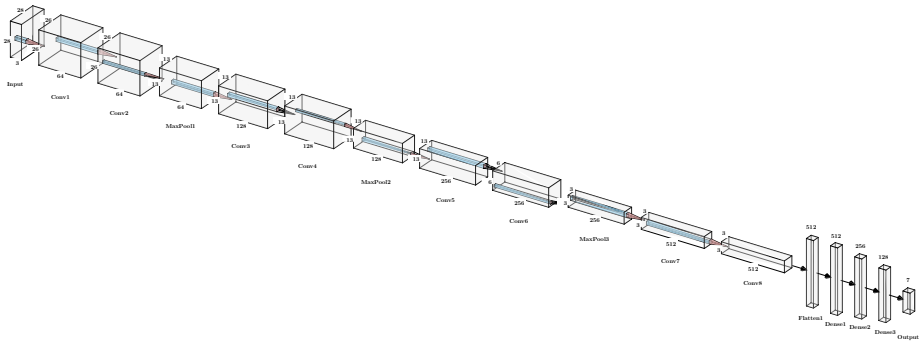

FIGURE 5.2: Diagram of `MyDeepCNN` architecture.

Since the training process with this deeper network was subsequently much longer than the shallow one, it was vital to use batch-normalisation for optimised gradient descents, as well, as some slightly different variables. For example, Adam was once again used as the optimiser, but with a `beta1` value of 0.9 and a new `beta2` value of 0.999. `L2` kernel regularisation was additionally employed on the first convolutional layer to prevent over-fitting.

As this network was substantially more complex than the previous shallow one, a higher-level and easier-to-use library called Keras was used instead of TensorFlow. This required numerous syntactic and logical changes in the code structure. However, the overall final code was much simpler to use, deploy and debug, despite being more complex than the shallower model

using TensorFlow. Indeed, the entire network was essentially built on only three simple methods: `create_model()`, `train()`, and `test()`. Unlike TensorFlow, Keras weights were not stored as `checkpoint` executables, but as `.h5` files.

TABLE 5.2: `MyDeepCNN` architecture table.

Layer	Size	Filter	Stride	# Filters	Activation
Input	28x28x3				
Conv1	26x26x64	3x3	1	64	ReLU
Conv2	26x26x64	3x3	1	64	ReLU
BatchNorm1	26x26x64				
MaxPool1	13x13x64	2x2	2		
Dropout1	13x13x64				
Conv3	13x13x128	3x3	1	128	ReLU
BatchNorm2	13x13x128				
Conv4	13x13x128	3x3	1	128	ReLU
BatchNorm3	13x13x128				
MaxPool2	6x6x128	2x2	2		
Dropout2	6x6x128				
Conv5	6x6x256	3x3	1	256	ReLU
BatchNorm4	6x6x256				
Conv6	6x6x256	3x3	1	256	ReLU
BatchNorm5	6x6x256				
MaxPool3	3x3x256	2x2	2		
Dropout3	3x3x256				
Conv7	3x3x512	3x3	1	512	ReLU
BatchNorm4	3x3x512				
Conv8	3x3x512	3x3	1	512	ReLU
BatchNorm5	3x3x512				
MaxPool4	1x1x512	2x2	2		
Dropout4	1x1x512				
Flatten1	512				
Dense1	512				ReLU
Dropout5	512				
Dense2	256				ReLU
Dropout6	256				
Dense3	128				ReLU
Dropout7	128				
Output	7				Softmax

5.2 Validation Methodology

The input data `x` and its corresponding labels `y` can be separated into training and testing sets. However, it is also important to have a *validation* set to measure the performance of the model *during* the training in order optimise it by tuning the hyper-parameters. The manner in which the validation set is taken from the main dataset can be done in a number of different ways, and which have interesting effects on the bias and variance trade-off on the measured accuracy. The subsections below analyse and explain the two validation approaches which were considered - the hold-out method and k-fold cross validation method - before identifying the approach that was ultimately selected and used in our models.

5.2.1 Hold-Out Validation

The *hold-out* validation methodology, also simply known as *validation set approach*, separates the entire initial dataset into three categories: training, validation and testing, as shown in Figure 5.3. The implemented method, available in the `MyUtilityMethods` class, can split the dataset in any preferred ratio, as shown n Listing 5.1.

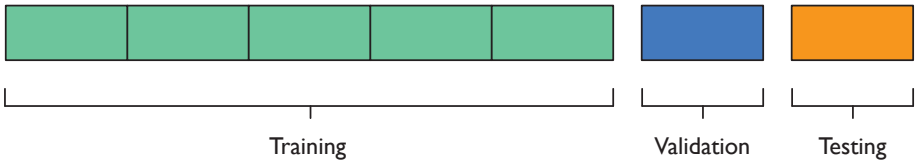

FIGURE 5.3: Distribution of data when using hold-out set validation.

```
1  def split_train_val_test(self, x, y, split1=0.9, split2=0.95):
2    '''Split dataset into train, validation, and test sets'''
3    x_train, x_val, x_test = np.split(x, [int(len(x)*split1),
     int(len(x)*split2)])
4    y_train, y_val, y_test = np.split(y, [int(len(y)*split1),
     int(len(y)*split2)])
5    return x_train, y_train, x_val, y_val, x_test, y_test
```

LISTING 5.1: Method employed to split dataset.

This is the standard and classic way of splitting the data, and the validation set serves as a constant metric to evaluate the performance accuracy of the model while it converges. While it is a simple approach from a conceptual and implementation point of view, it has a number of drawbacks from a bias-variance analysis perspective. For example, the accuracy and the loss can both be highly *variable*, as they simply depend on the split chosen for the validation set from the overall dataset. Indeed, as the validation set contains a limited and constant number of total observations, the accuracy

and loss can vary substantially depending simply on the validation samples chosen.

Additionally, having a fixed validation set, i.e. the same validation samples at each epoch, implies that the training data also remains constant, as the latter is simply the remaining data. Since a model can only train and fit on the training dataset itself, its performance will *decrease* as the validation set size increases, because the latter, in turn, decreases the training set size. Statistical modelling techniques yield lower overall accuracy when trained on fewer samples, which suggests that the validation accuracy and loss might actually *overestimate* the eventual test accuracy and loss, when fitting the entire dataset.

5.2.2 k-Fold Cross Validation

A reasonable alternative to the hold-out validation methodology is the k-fold cross validation method. This method divides the entire training dataset into k groups, or *folds*, out of which $k-1$ are used as observations samples for training, and 1 is used as the validation set. As we train through all the epochs, the validation set is alternated between the k groups, which eventually allows the model to fit over the *entire dataset* over the course of the training, as depicted in Figure 5.4.

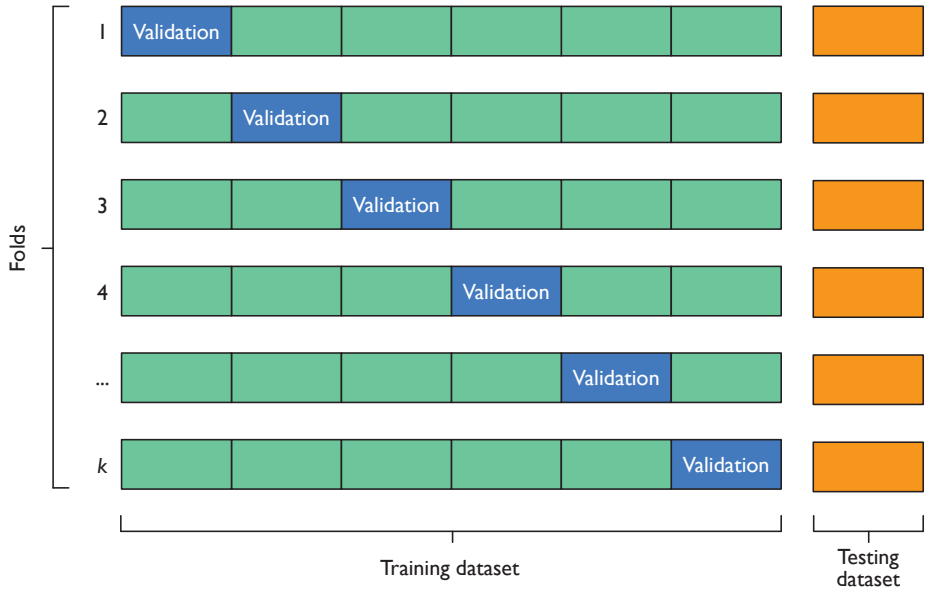

FIGURE 5.4: Distribution of data when using k-fold cross validation.

This method *reduces* the validation variability by a significant magnitude when compared to the hold-out method, and in turn increases the accuracy of estimates of the test losses. The k-fold cross validation approach also

provides a lower *bias* than the hold-out method, as each training set contains $(k-1)n/k$ samples, which is generally more than the samples in the hold-out method, depending on the `split` parameter and the value of k. As summarised by James et al. (2014), it has been empirically shown that the optimal value of k for k-fold cross validation are $k = 5$ or $k = 10$, as these tend not to have a high bias or an excessive variance.

The networks `MyCNN` and `MyDeepCNN` both purposefully allow for *either* methodology to be used when training a model, as they were engineered using the design principles outlined in chapter 4. No external library was used for the development of this feature, as it was implemented in standard NumPy, and simply required a certain number of shape manipulations in the training parameter variables. We can thus choose to experiment, both with and without cross validation, to see if the theoretical results actually correspond to those in practice, for the various tasks.

Furthermore, the value of k was selected to be equal to the *number of epochs*, as opposed to $k = 5$ or $k = 10$, as it was a more interesting experiment. This implied that no validation fold was ever repeated, as k was never smaller than `nb_epochs`, and thus divided into exactly k *unique* groups, one for each epoch of the training phase. This essentially turns the `nb_epochs` variable from a standard parameter into an important hyper-parameter which has a significant influence on the bias-variance trade-off when using the k-fold cross validation methodology. This allows us to experiment directly on this variable and see how well the final results are correlated to the selected variables and their predicted theoretical behaviour.

5.3 Metrics

Although one might develop various architectures, use different validation methodologies, and develop an ample number of experiments and hypotheses, in order to effectively compare and evaluate them with one another we need a comprehensive benchmarking system with common measurement criteria that can provide feedback on all the different models. The following paragraphs give an overview of the benchmarking methods used to measure the performance of the various models, which are included in the `My Utility Methods` class.

Since the three main problems to be solved are all multi-classification tasks, the criteria and evaluation methodology is fundamentally the same, and can be developed with relative ease. The first two principal metrics are the loss and accuracy metrics. Both are measured during the training phase against

the training and validation labels, but the latter is especially relevant when measuring against the testing dataset. This is indispensable for assessing whether a model converged during training or not, as they provide metrics which evolve through the course of the epochs. These *learning curves* allow one to perform the very initial analysis - determining if the model is over - or under-fitting - which also gives an indication of the bias-variance trade-off. Based on observations, one can tune various hyper-parameters to optimise the model and improve its performance.

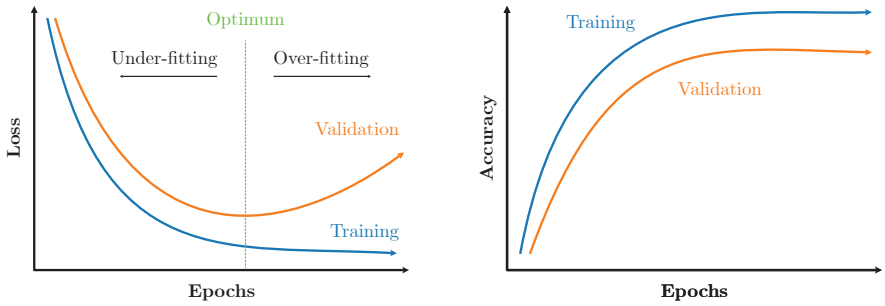

FIGURE 5.5: Conceptual loss and accuracy learning curves in function of epochs during training. The optimal moment to stop the training is given in green.

However, much more detailed metrics are required after a certain point in order to further tweak our system effectively. This is when confusion matrices come into play. The overall mean accuracy of a model reveals no information about the correct classification rates *per class*. A classification model can make two types of errors: it can either correctly classify a sample in its actual class, or it can incorrectly predict it to be in another class. A confusion matrix allows one to visualise the accuracy of a model's classifier by comparing its predictions against the ground truths for each class.

As shown in Figure 5.6, we can calculate the true positives (TP), true negatives (TN), as well as, false positives (FP) and false negatives (FN), for each class. This in turn, allows us to derive additional metrics such as the true positive rate (TPR), false positive rate (FPR), and positive predictive value (PPV). These are also known as the recall, fall-out, and precision rates, and can be accurately used to measure a classifier. However, when working with multiple models, a single, final criteria is often required to concretely rank their overall performance. The F1-Score is one of the possible metrics for this scenario, as it gives the harmonic mean of the precision and sensitivity.

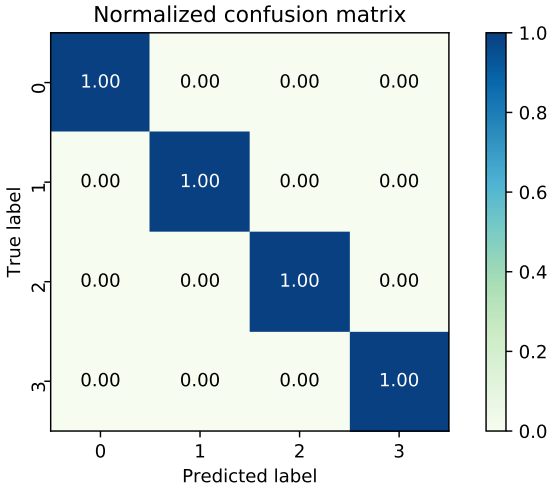

FIGURE 5.6: Ideal confusion matrix of 4 classes. The diagonal and off-diagonal cells represent the model's normalised correct and incorrect class predictions rates respectively.

$$\text{Recall} = \text{TPR} = \text{TP}/(\text{TP}+\text{FN}) \tag{5.1}$$

$$\text{Fall-out} = \text{FPR} = \text{FP}/(\text{FP}+\text{TN}) \tag{5.2}$$

$$\text{Precision} = \text{PPV} = \text{TP}/(\text{TP}+\text{FP}) \tag{5.3}$$

$$\text{F1 Score} = (\text{TPR} \cdot \text{PPV})/(\text{TPR} + \text{PPV}) \tag{5.4}$$

In addition, we can also use the TPR and TNR to effectively visualise these evaluation metrics on a graph. The Receiver Operating Characteristic (ROC) curve is a popular plot which illustrates the diagnostic ability of a binary classifier system by simultaneously displaying type I and II errors. It can be visualised by plotting a classifier's TRP against its FPR, as shown in Figure 5.7.

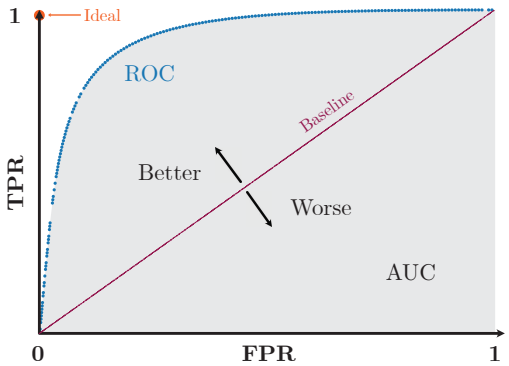

FIGURE 5.7: Sample ROC curve and its corresponding Area Under the Curve (AUC). The diagonal baseline represents a 'line of no-discrimination', and the (0,1) spot constitutes the perfect classification point.

This only works for binary classifications problems, such as the gender classification task, but can be easily extended to multi-class problems by measuring the rates with a 'one vs. rest' methodology, and then plotting the individual ROC curves for each class. We can then average all the curves to calculate and plot the *macro-average* ROC curve, which gives equal weight to the classification of each class. Moreover, we can also calculate and draw the *micro-average* ROC curve by considering each element of the class indicator matrix as a binary prediction.

Finally, an alternative to the F1-score as the overall performance of a classifier is the Area Under the Curve (AUC) of the ROC curves. It essentially gives a number to the ROC curves, which tell us the strength of classification rates in numbers. We ideally want all our lines to be as close as possible to the ideal (0,1) point, and thus the AUC to be as close to 1 as possible.

5.4 Models

Once a comprehensive metrics system which could cover any type of classification problem and both network architecture types was established, one had to finally decide on the models for each task. A number of different models were considered for each task, and the ones listed below in Table 5.3 were the models that were implemented.

Table 5.3: List of the models developed per task.

Task	Type	Validation	Activation	Library
Face Recognition	`MyCNN`	Hold-out	Softmax	TF `1.14.0`
Face Recognition	`MyCNN`	k-Fold	Softmax	TF `1.14.0`
Gender Classification	`MyCNN`	k-Fold	Softmax	TF `1.14.0`
Gender Classification	`MyCNN`	k-Fold	Sigmoid	TF `1.14.0`
Emotion Detection	`MyCNN`	Hold-out	Softmax	TF `1.14.0`
Emotion Detection	`MyCNN`	k-Fold	Softmax	TF `1.14.0`
Emotion Detection	`MyDeepCNN`	Hold-out	Softmax	Keras `2.2.4`

As the facial recognition task was a multi-class problem, which in theory was directly solvable by the shallow `MyCNN` network, we decided use it and experiment on its validation methodology, to see if either type would produce tangible differences on our data.

The gender classification problem was a binary classification problem, which is why we used the opportunity to experiment with the activation layer and

see if there was a substantial difference in the accuracies of a softmax or sigmoid-activated approaches to the models. To this end, we kept the same validation methodology as we did not want any other factor to interfere in our experiment and its subsequent analysis.

Finally, the emotion detection task was deemed to be significantly more challenging and complex, which is why we decided to try both the shallow and the deep `MyDeepCNN` network on it, to analyse how well model complexity correlates with the apparent difficulty of a problem. We also wanted to observe if the advanced machine learning techniques such as dropouts, batch-normalisation, and `l2` regularisation would considerably change the final overall accuracy. The shallow network was evaluated on both validation methodologies, but only with a softmax activation function as it was a non-binary, multi-class problem.

In the next chapter, we train the models shown in Table 5.3, and attempt to gain insights into the individual problems of the various models, by analysing the results and metric evaluations.

Chapter 6

Results and Analysis

This chapter presents the hyper-parameters chosen for each model, as well as, the results of the training, including evaluation metrics scores and the corresponding plots.

6.1 Face Recognition

6.1.1 Hold-out Model

TABLE 6.1: Hyper-parameters choices.

Hyper-parameter	Value
Network	`MyCNN`
Validation type	Hold-out
Number of epochs	50
Learning Rate	1×10^{-5}
Batch size	30
Final activation function	Softmax
Random seed	0

The hyper-parameters chosen for the hold-out model in order to solve the facial recognition task are given in Table 6.1. These were the optimal values found after manual tuning, and which helped produce the excellent learning curves shown in Figure 6.1.

One can see that both, the given accuracy and loss graphs, correspond very closely to the conceptual graph in Figure 5.5, previously shown in chapter 5. Fifty epochs were sufficient for the model to train well without over-fitting or losing the ability to generalise over new data.

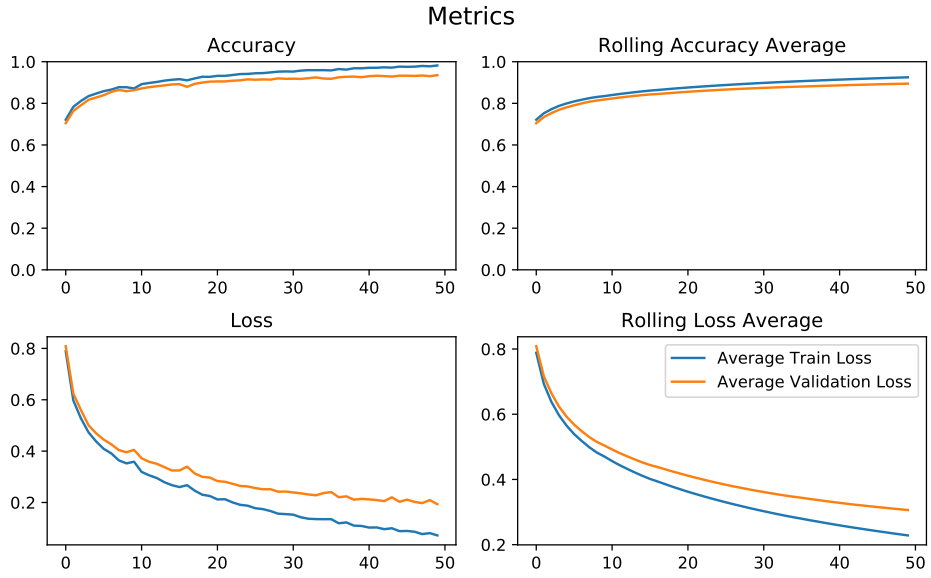

FIGURE 6.1: Training loss and accuracy evolution.

Furthermore, as seen in the confusion matrices given in Figure 6.2, the model performs well not only on the training and validation dataset, but is also able to fit unseen data remarkably well. Indeed, the overall *testing* dataset accuracy was **95.1%**.

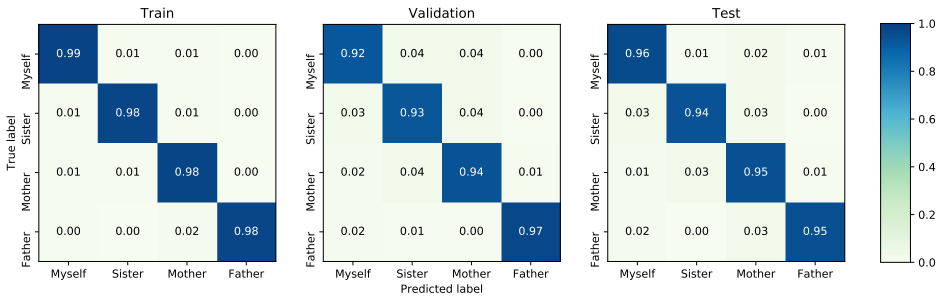

FIGURE 6.2: Confusion matrices for all three datasets.

Using the true and false positive rates calculated from the confusion matrices, we can plot the ROC curves for each class. Figure 6.3 gives the testing ROC curves, along with its rounded-off AUC values. The corresponding training ROC curves can also be seen in Figure B.1. We can observe that all classes perform well, with the 'Myself' class slightly out-performing all the others.

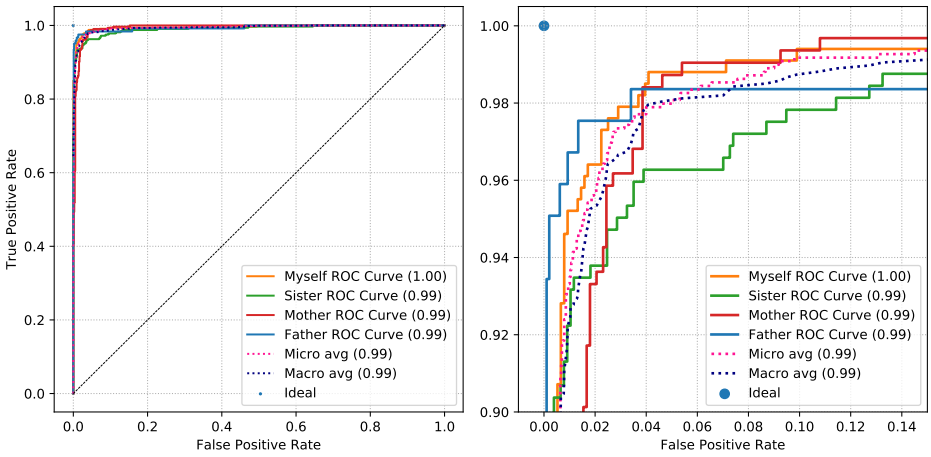

FIGURE 6.3: Testing dataset ROC curves per class.

Additional numerical metrics are given in Table C.1, C.2, and C.3, which include the precision, recall, and in particular, the F1-scores of each class for the training, validation, and testing dataset. In this case, they actually serve as a better benchmark criteria and reference against one another than the AUC values, as these were much more homogeneous.

6.1.2 K-fold Model

When training with the k-fold validation methodology, the hyper-parameters did not require to be changed from the hold-out model at all, except for the final activation function, as shown in Table A.2. We can observe that the learning curves shown in Figure 6.4 are somewhat similar to the hold-out model. In particular, the validation set's accuracy and loss both have higher variance.

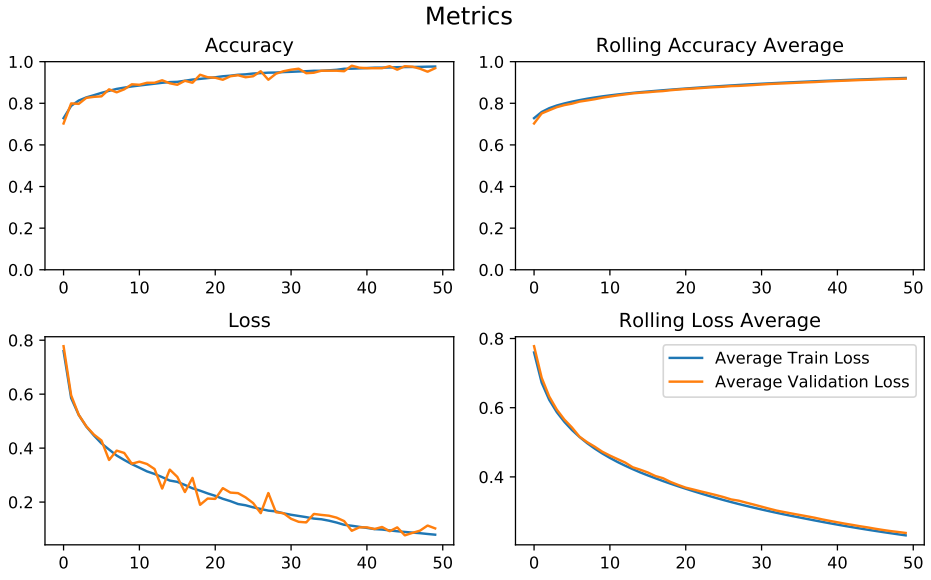

FIGURE 6.4: Training loss and accuracy evolution.

The confusion matrices given in Figure 6.5 are also similar to the hold-out model, with both achieving >94% accuracy. The overall testing accuracy for this model is **94.5%**.

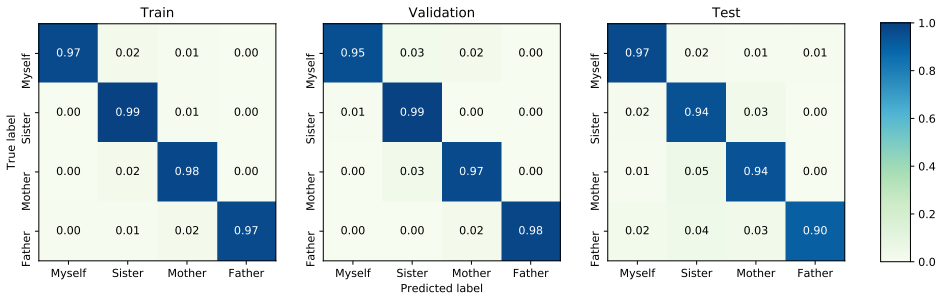

FIGURE 6.5: Confusion matrices for all three datasets.

The ROC curves given in Figure 6.6, and their corresponding AUCs are nearly identical to the hold-out set methodology, with the 'Myself' class again performing slightly better than the rest in the testing dataset. The remaining numerical metrics are given in Table C.4, C.5, and C.6.

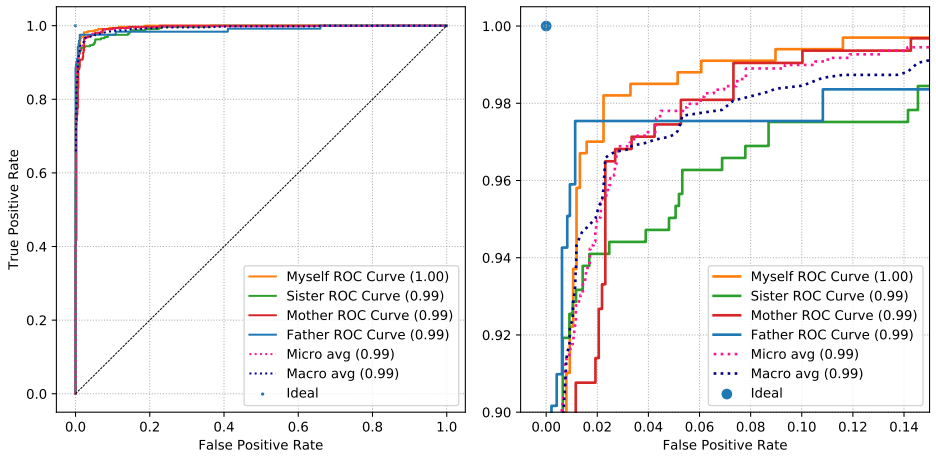

FIGURE 6.6: Testing dataset ROC curves per class.

6.1.3 Comparing Models

We can compare both models by evaluating two conclusive metrics given previously, namely the F1-score or the Area Under the Curve (AUC). To this end, we plot separate ROC curves for each class by methodology type, as shown in Figure 6.7. We can observe that the AUCs are nearly identical for all classes, except for the macro average. Table 6.2 shows that the hold-out method has a slightly higher average F1-score over all the classes.

We can thus conclude that this task was successfully solved using our own dataset, with two different methodologies. These do not result in significant

differences with the chosen hyper-parameters and are able to generalise very well on unseen testing data, averaging an accuracy of approximately 95%.

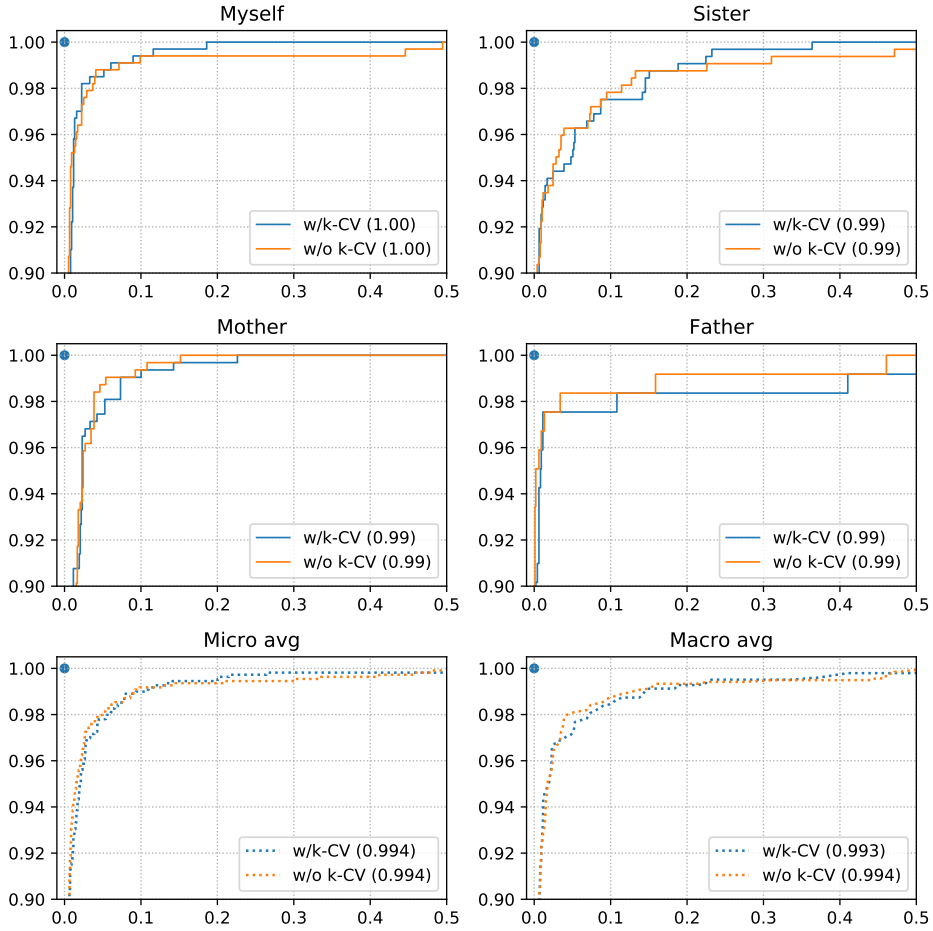

FIGURE 6.7: ROC curves and their respective AUCs comparing hold-out and k-fold cross validation methodology for each class.

TABLE 6.2: F1 scores per validation methodology.

	Hold-out	k-Fold
Myself	0.96	0.97
Sister	0.95	0.93
Mother	0.94	0.94
Father	0.95	0.93
Average	0.95	0.94

6.2 Gender Classification

For the gender classification task, we train both models with k-fold cross validation. Then, we experiment with the final activation layer, alternating

between a softmax and a sigmoid function. The following sections give the results for these two models and compares their metrics.

6.2.1 Softmax Model

The learning rate and other hyper-parameters, given in Table A.3, remain the same as the previous facial recognition task.

We can observe that the learning curves shown in Figure 6.8 is similar to the k-fold cross validation model for the facial recognition task, with a high variance on the validation dataset for both loss and accuracy. Nonetheless, the model converges at a final testing accuracy of **97.0%**.

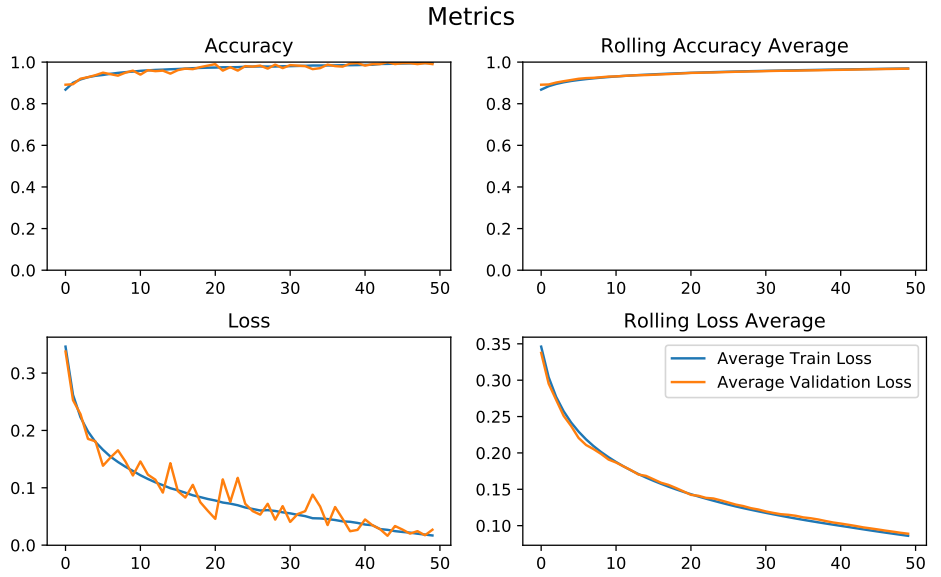

FIGURE 6.8: Accuracy and loss learning curves.

As expected from such a result, the confusion matrices shown in Figure 6.9 reflect similar high scores for all the datasets, with a slightly higher Type I error on the testing dataset.

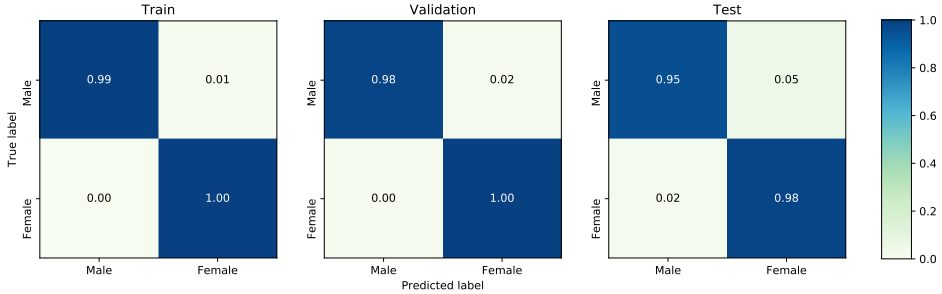

FIGURE 6.9: Confusion matrices for all three datasets.

We can use the true and false positive rates to plot the ROC curves for each class, and their corresponding Area Under the Curves for the training and

testing datasets. The 'Male' class seems to outperform the 'Female' class in this criteria by a negligible margin - both have very high scores at $>$**99%**. On the other hand, we can observe in Table C.9 that 'Female' outperforms the other class in F1-scores, again by a small margin. The corresponding train and validation score tables are given in Table C.7 and Table C.8.

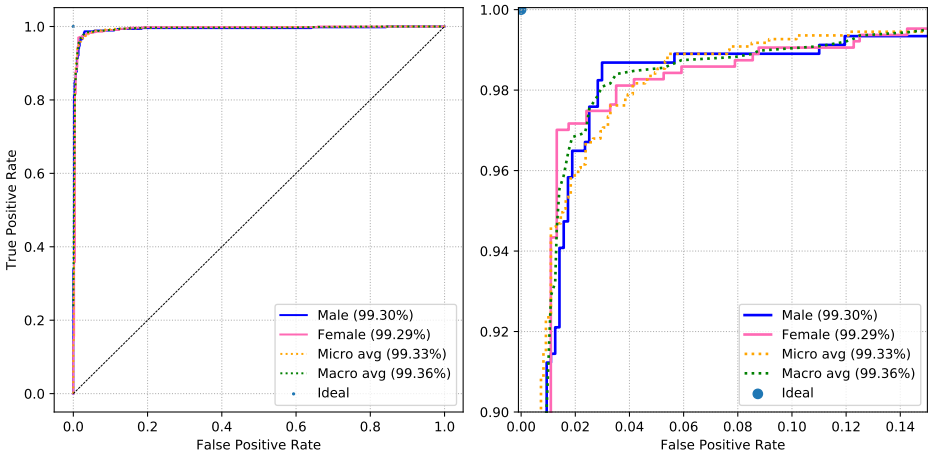

FIGURE 6.10: Testing dataset ROC curves per class.

6.2.2 Sigmoid Model

We change the final activation function to sigmoid but leave all the remaining hyper-parameters as before. The learning curves and confusion matrices are nearly identical to the softmax model, shown in Figure 6.11 and Figure 6.12. The training and testing dataset ROC curves are again given in Figure B.7 and Figure 6.13, and additional numerical metrics can be seen in Table C.10, Table C.11, and Table C.12.

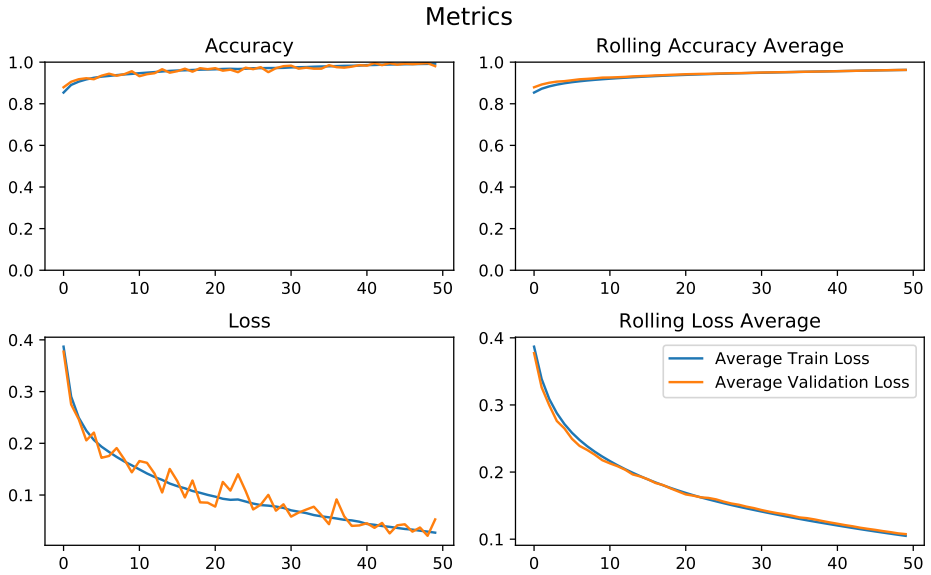

FIGURE 6.11: Training loss and accuracy evolution.

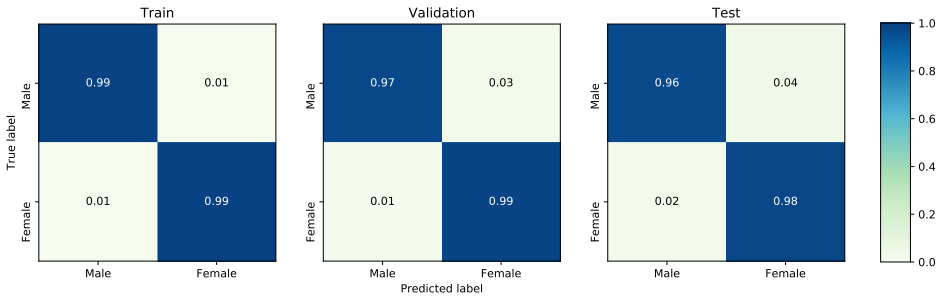

FIGURE 6.12: Confusion matrices for all three datasets.

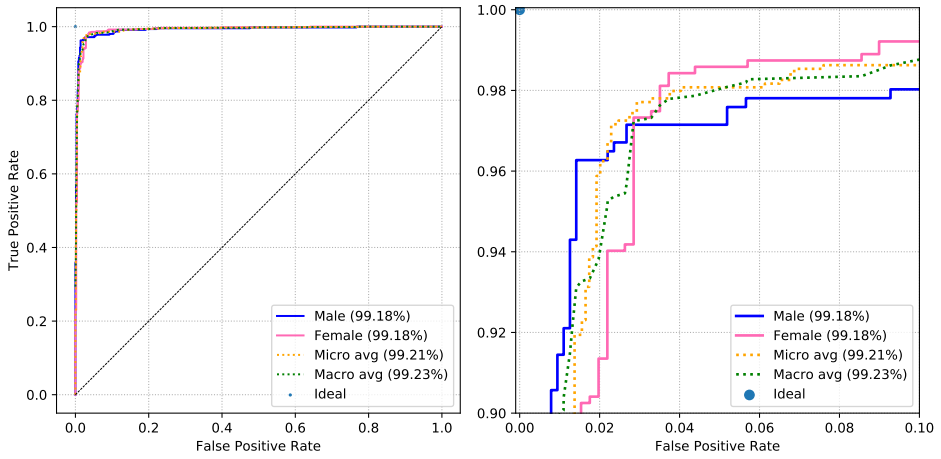

FIGURE 6.13: Testing dataset ROC curves per class.

6.2.3 Compare Models

We evaluate the two models by again comparing their ROC curves by plotting per class but with both models. We can observe in Figure 6.14, that the softmax models slightly outperform the sigmoid model for both classes, which also naturally reflects in the micro and macro-average. When comparing their F1-scores as in Table 6.3, we can observe that they average out exactly to the same value.

We thus conclude this task to be very successful, as it yields the highest overall scores of all tasks, with either activation function.

TABLE 6.3: F1 scores of each model.

	Softmax	Sigmoid
Male	0.97	0.97
Female	0.98	0.98
Average	0.98	0.98

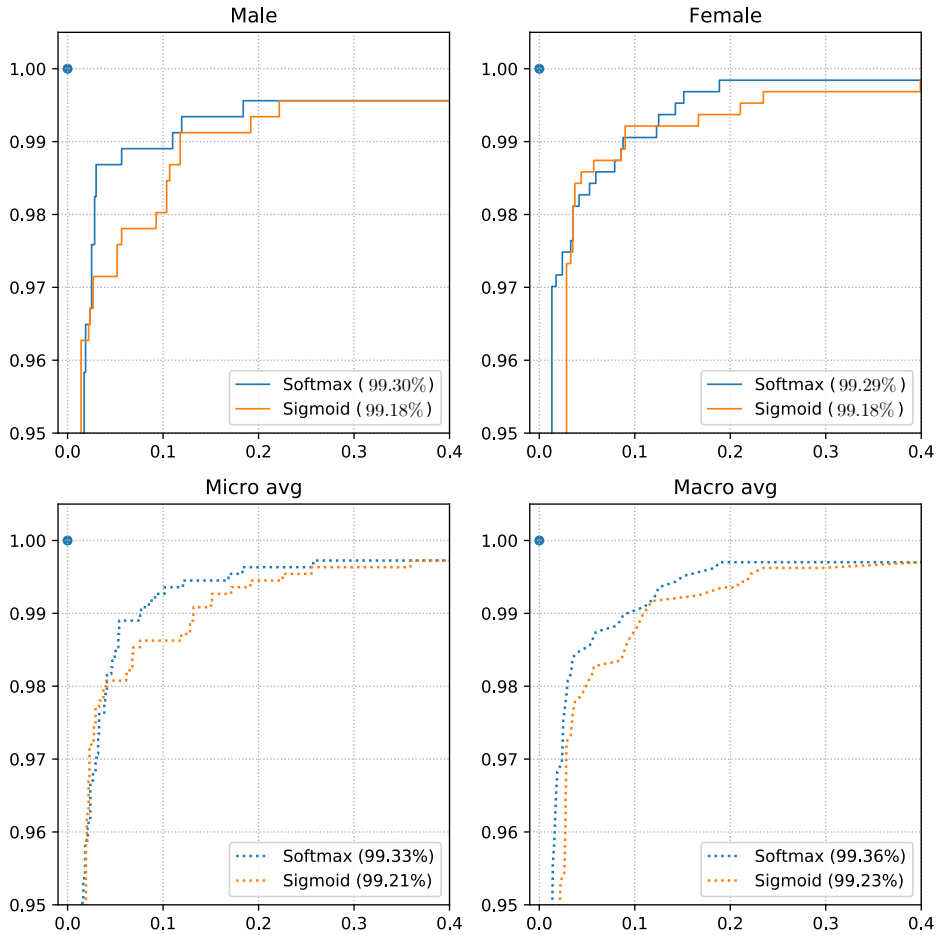

FIGURE 6.14: ROC curves and their respective AUCs comparing sigmoid and softmax models for each class.

6.3 Emotion Detection

Finally, for the last task, we implement three different models: two shallow `MyCNN` networks, with k-fold cross validation and hold-out set methodology; and one with deep `MyDeepCNN`. The sections below summarise the interesting results for this task.

6.3.1 Hold-out Model

The first implemented model to solve the emotion detection task used a hold-out set validation methodology, softmax activation function, and a learning rate smaller than the previous models, as given in Table A.5. Although the remaining hyper-parameters were the same as the previous models, the learning accuracy and loss curves during training were not optimal at all. Indeed, as observed by the loss graph in Figure 6.15, the model clearly over-fits, as shown by the conceptual graph given in Figure 5.5. The loss clearly goes beyond the 'optimal point', which in this case, actually

does not even ever appear. The validation accuracy stagnates at around only 40%. The final testing dataset's accuracy was also only **47.7%**.

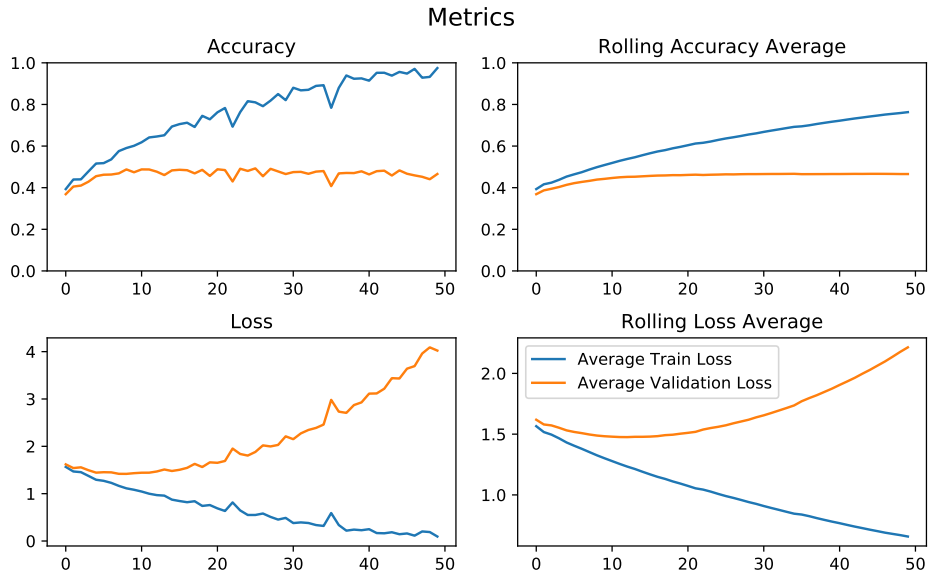

FIGURE 6.15: Training loss and accuracy evolution.

The training dataset's confusion matrix yields good results, but does very poorly on the validation and test sets, which confirms the model's inability to generalise over new data. Indeed, it confirms that the model over-fitted and memorised the training dataset.

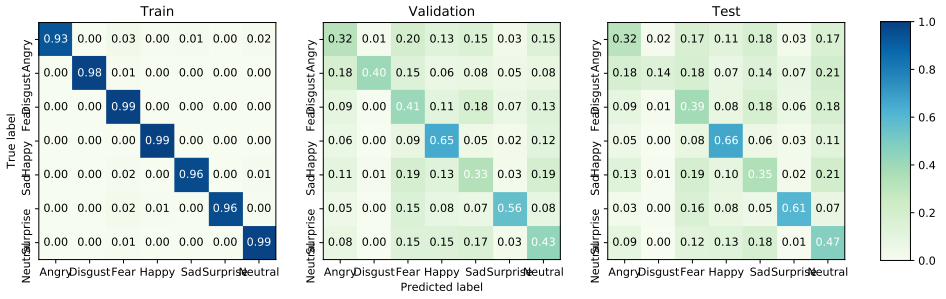

FIGURE 6.16: Confusion matrices for all three datasets.

This is further reflected in the class ROC curves, which are close to the ideal (0,1) spot in the training dataset, but do not even fit in the same axes scale for the testing dataset, as shown in Figure B.9 and Figure 6.17 respectively. Furthermore, the Areas Under the Curve macro-average is only at **77.95%** for the testing set, by far the lowest score for any model over all tasks combined. Additional numeric metrics can be seen in Table C.13, Table C.14, and Table C.15.

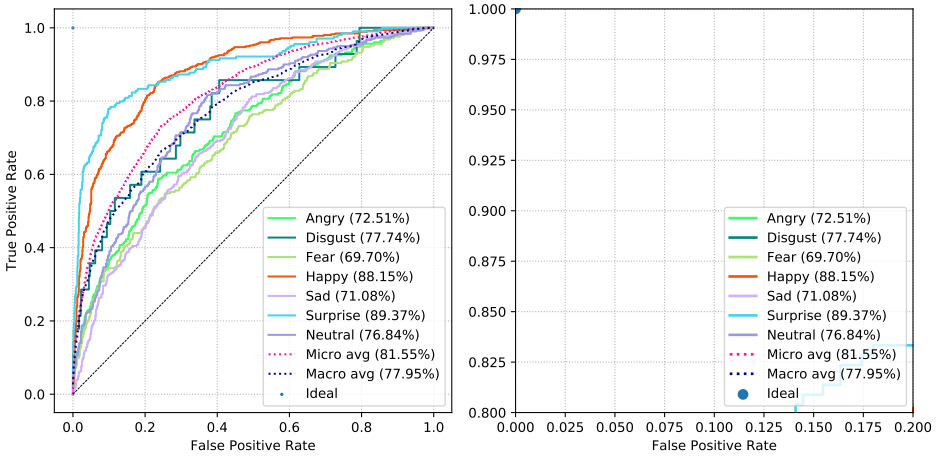

FIGURE 6.17: Testing dataset ROC curves per class. The zoomed-in version on the right-hand side did not even fit in the given scales of the axes.

6.3.2 K-Fold Model

The k-fold cross validation model only changes the validation type as the hyper-parameter, with the rest remaining the same as the hold-out model. The learning curves shown in Figure 6.18 are quite different from all the previous models, as this does not seem to converge in the given epochs. The validation dataset accuracy and loss have a high variance as expected, but neither curves level out during training.

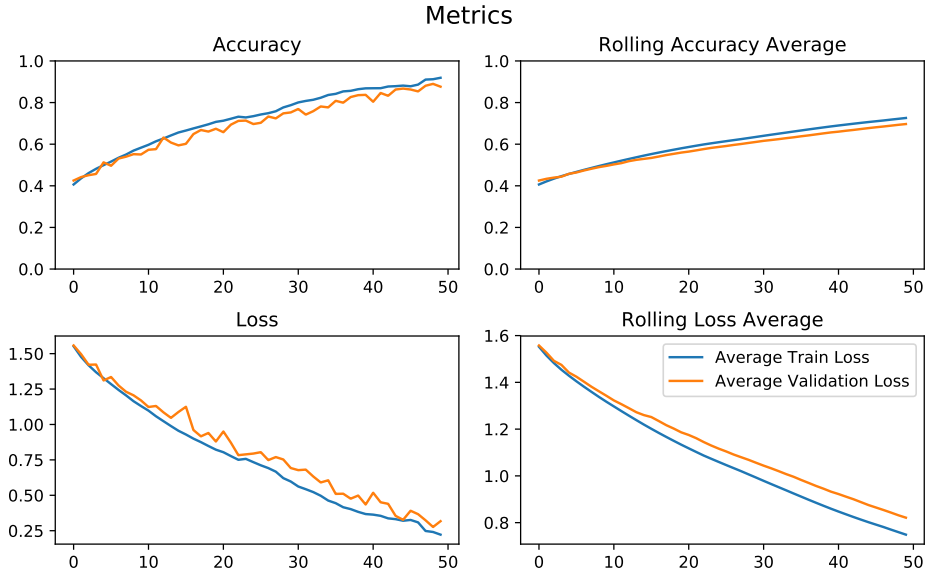

FIGURE 6.18: Accuracy and loss learning curves.

When observing the confusion matrices, shown in Figure 6.19, we can observe an interesting phenomenon: the model, this time, performs significantly better at the validation dataset, but still fails to have good overall results on the testing dataset. A deeper analysis suggests that the model is

in fact no better than the hold-out model, and that the validation confusion matrix further demonstrates that this model (along with the previous one) over-fits.

Indeed, since the model tends to memorise the training dataset when over-fitting, and because the validation dataset in the k-fold cross validation methodology takes alternating group samples from the training dataset itself, it is very likely that the model simply learnt to memorise over those samples in question as well. These were not withheld from the model from the start as in the hold-out model, and thus provide a definitive reason as to why the confusion matrix gives good results for both the training and validation dataset, but severely under-performs when working with the testing dataset.

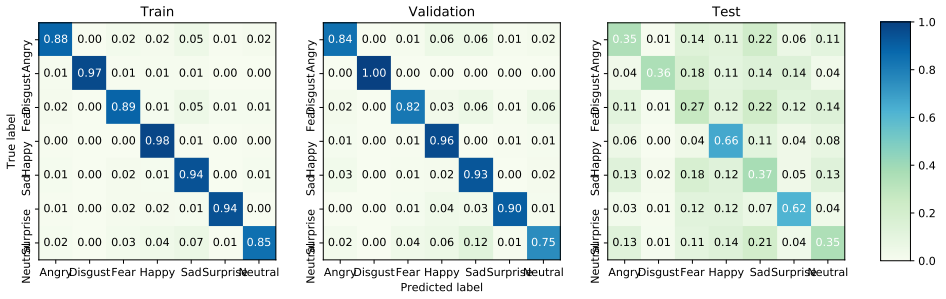

FIGURE 6.19: Confusion matrices for all three datasets.

We can observe the macro-average for the testing dataset's AUC is again at only **77.20%**, nearly identical as for the hold-out model. Therefore we chose to further experiment on this problem with a deeper model to see if it manages to converge.

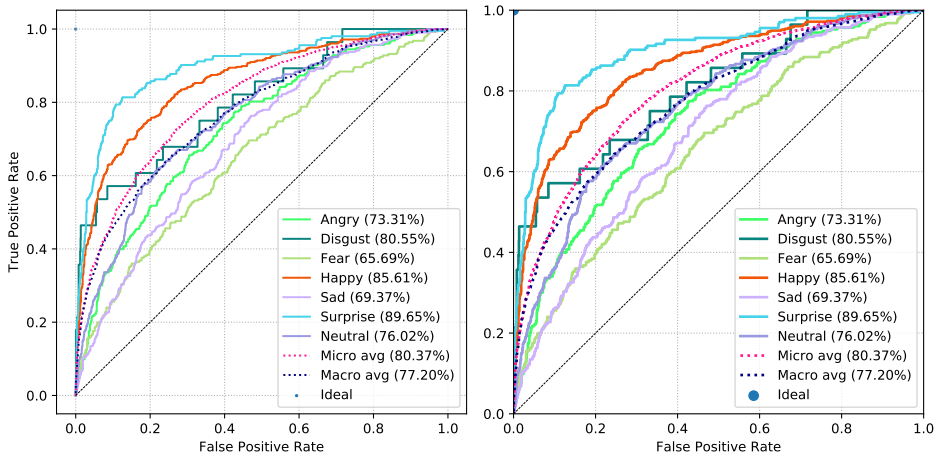

FIGURE 6.20: Testing dataset ROC curves per class.

6.3.3 Deep Network Model

The deep model has slightly more nuanced hyper-parameters, given in detail in Table 6.4. It uses a hold-out validation methodology, and a learning decaying specified by the two `beta1` and `beta2` parameter values. Furthermore, a training batch size of 32 is also chosen.

TABLE 6.4: Deep Network Model

Hyper-parameter	Value
Network	`MyDeepCNN`
Validation type	Hold-out
Number of epochs	50
Learning Rate	1×10^{-6}
Batch size	32
Final activation function	Softmax
Random seed	0
Beta 1	1×10^{-6}
Beta 2	1×10^{-6}
Epsilon	1×10^{-6}
Number of features	32

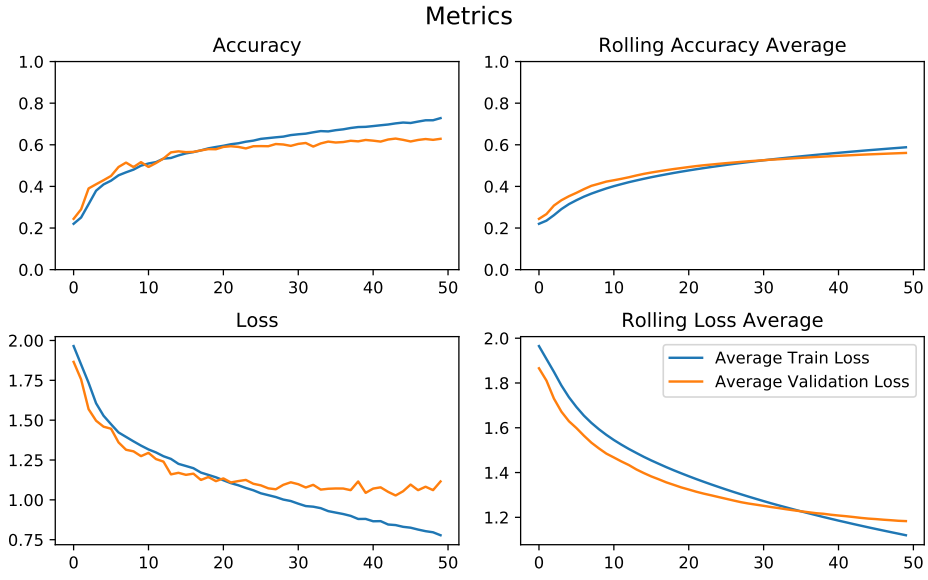

FIGURE 6.21: Deep network's learning curves.

We can see from this model's learning curves, shown in Figure 6.21, that the model successfully converges. The loss and accuracy both even out, the latter at approximately 60%, confirmed by the overall final testing accuracy of **62.5%**. Although this value is lower than any of the previous (working)

models, it is still a substantive result considering the highest obtained accuracy for the FER2013 dataset is only **71%**.

We can indeed confirm the model's ability to generalise over unseen data, by the testing dataset's confusion matrix, shown in Figure 6.22. It is finally able to successfully produce good performance, and does not differ from the validation set's matrix, unlike the previous model.

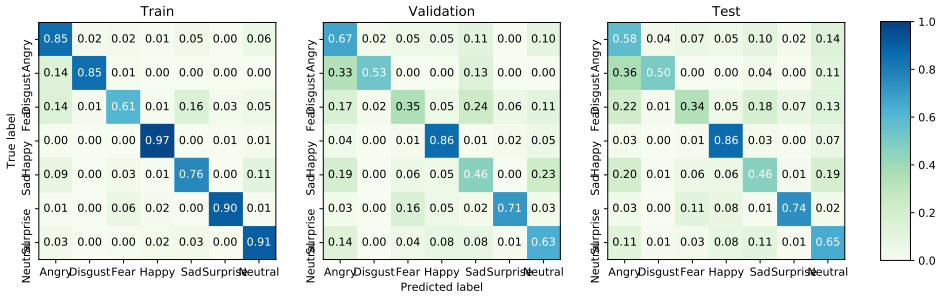

FIGURE 6.22: Deep network's confusion matrices.

The testing ROC curves are correspondingly better, with a macro-average of **88.45%**, a substantial increase over the previous two shallow models.

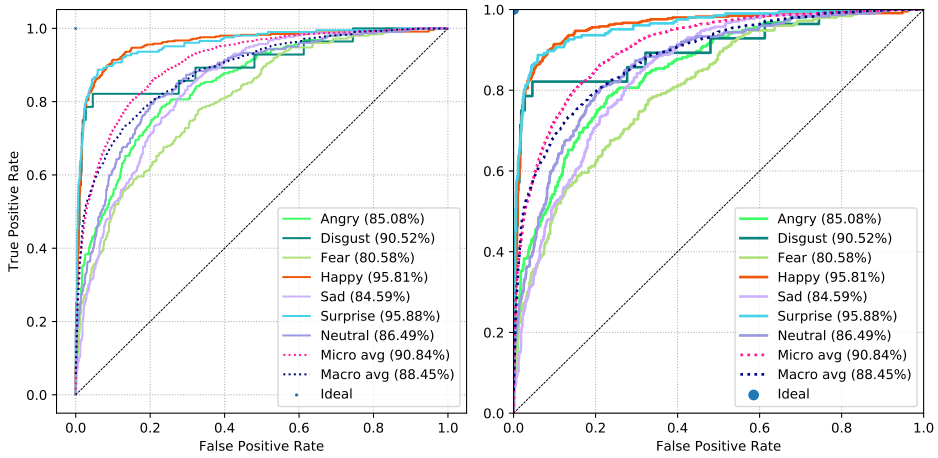

FIGURE 6.23: Testing dataset ROC curves per class.

Further numeric metrics can be seen in the tables given in Table C.19, Table C.20, and Table C.21.

6.3.4 Compare Models

We can now compare all three models based on the Area Under the Curve and F1-scores. We can see an improvement in all AUC scores when switching from the shallow models to the deeper model. The final macro average of the deep model is **88%**.

When comparing the F1-scores, we can clearly see the hold-out and k-fold cross validation models average at the same score of 0.43 and 0.42 respectively, however the deep network is significantly better at **0.58**.

Table 6.5: AUCs and F1-scores of all models.

	AUC			**F1-Score**		
	Hold-out	k-Fold	Deep	Hold-out	k-Fold	Deep
Angry	0.73	0.73	0.85	0.36	0.37	0.51
Disgust	0.77	0.80	0.91	0.19	0.36	0.45
Fear	0.70	0.66	0.81	0.35	0.27	0.41
Happy	0.88	0.86	0.95	0.68	0.65	0.84
Sad	0.71	0.69	0.85	0.35	0.34	0.50
Surprise	0.89	0.90	0.96	0.66	0.59	0.78
Neutral	0.77	0.76	0.86	0.43	0.38	0.59
Micro avg	0.82	0.80	0.91	-	-	-
Macro avg	0.78	0.77	0.88	-	-	-
Average	0.78	0.78	0.88	0.43	0.42	0.58

With the deep model's results, we can finally conclude to have solved the 'emotion detection' problem ; and even though the final testing accuracy is not in the same league as for the 'face recognition' and 'gender classification' problems, it is important to recall and remind ourselves of the very complex nature of this particular task.

6.4 Conclusion

In this chapter we summarised the successful implementation of multiple models with different hyper-parameters and validation methodologies for each task, with results ranging from reasonable to excellent.

We have shown that in the context of facial recognition tasks, the hold-out and k-fold cross validation methodologies do not result in significantly different results, as per our evaluation metrics. Indeed, the former methodology interestingly gives the validation learning curve a higher variance, but does not substantially differ in F1-scores or Area Under the Curve values.

We have also solved the gender classification task using two different activation functions, both of which perform exceptionally well in all metrics, and do not significantly diverge from one another to be conclusive in relation to the two approaches used.

The emotion classification task yields the most interesting results for analysis. Unlike the previous two tasks, which were solved with shallow networks, this problem proved to be solvable only by using a much deeper network. Indeed, both the shallow hold-out and k-fold cross validation models tended to over-fit the data, and in fact did not converge on the testing data at all, as clearly seen on the corresponding confusion matrices and ROC curves.

The deep network model, however, proved to be robust enough to converge to a reasonable accuracy, which was significantly higher than the other two models. The reason for its success can be confidently attributed to the higher number of layers, as well as, the other relatively novel deep learning techniques that were used, such as batch-normalisation and dropout units, which seem to be necessary for solving problems of considerably higher level of complexity.

Based on the above evidence and learning, in the next chapter 7, we discuss the end-to-end model where we attempted to combine all three tasks to work seamlessly together as a single unified model.

Chapter 7

End-to-End Model

The results obtained from the individual models were very satisfactory, with each model achieving a high level of performance, as demonstrated through various performance metrics. However, our goal was to ultimately have all of them work together seamlessly in a combined *end-to-end* model. This meant taking in any image as input and being able to directly perform all three tasks - facial recognition, gender classification, and emotion detection - within a unified model. To that end, we built a final Python script which could use the trained and frozen weights of all trained models and run them for testing on completely unseen data. The following sections summarise the process which went into building this script.

7.1 Engineering

As all the models had been fully trained, and then further optimised with hyper-parameter tuning, we did not want to further update the weights. We loaded the saved weights, `checkpoint` executable files for TensorFlow, and `.h5` files for Keras, from the correct directory into our script, as shown in Listing 7.1. This creates all the necessary variables such as `tf.Session` and `saver`, by loading the saved graph and restoring the latest checkpoint. This also allows us to retrieve the last layer's output softmax tensor by their saved name, e.g. `CNN/Softmax:0`.

```
1  def restore_model(graph, graph_dir, checkpoint_dir):
2      '''Import tf graph and restore model variables'''
3      with graph.as_default():
4
5          # Load graph and restore tf variables
6          saver = tf.train.import_meta_graph(graph_dir)
7          latest_checkpoint = tf.train.latest_checkpoint(
       checkpoint_dir)
8          sess = tf.Session(graph=graph)
9          saver.restore(sess, latest_checkpoint)
10
11         # Get relevant tensors
12         tf_cnn_softmax=graph.get_tensor_by_name('CNN/Softmax:0')
```

```
13          tf_placeholder=graph.get_tensor_by_name('Placeholder:0')
14
15      return sess, tf_cnn_softmax, tf_placeholder
```

Listing 7.1: Restore trained weights.

The input data then needed to be pre-processed in exactly the same way as during training, i.e. with the help of histogram equalisation, standardisation, re-sizing, etc. `OpenCV` was again used to detect the faces on the image using Haar's Cascade.

The difficulty in this model was in correctly initialising all TensorFlow variables, sometimes multiple ones for each task. Once these were correctly set up, and the two tensors - `tf_cnn_softmax` and `tf_placeholder` - obtained, we could easily run inference with the `sess.run()` or `model.predict()` method, as shown in Listing 7.2.

```
1 def run_model(sess, tf_placeholder, tf_cnn_softmax, x_test):
2     '''Run model'''
3     probs = sess.run(tf_cnn_softmax, feed_dict={tf_placeholder:
      x_test})
4     y_hat = np.argmax(probs, axis=1)
5     return probs, y_hat
```

Listing 7.2: Run inference on restored weights.

This process returns the probabilities and prediction classes for the input testing data, which we can then use to present the results directly on the images. Since this script mixed all three tasks together, it was important to be able to convey a lot of visual information at once, and not unnecessarily overcrowd images with text giving the prediction names. Therefore, we decided to convey the results of each task in a different way.

For example, the gender classification predictions were given through different **shapes** around the face in question - *rectangles* for males and *circles* for females. The facial recognition task was conveyed through the **colour** of the shape, with each colour corresponding to the core variable defined in chapter 4, e.g. *orange* for 'Myself', *green* for 'Sister', *red* for 'Mother', and *blue* for 'Father'.

Finally, emotion was displayed through **text** on the top left-hand corner of side respectively, of the rectangle or circle circumscribing the faces. We also decided to include the corresponding *probability* of the emotion, as this was a complex task with some level of uncertainty. All of these mark-up tasks were easily achieved with `OpenCV`'s methods, such as `cv2.rectangle()`, `cv2.circle()`, and `cv2.putText()`.

7.2 Results

The results of the end-to-end model can be seen on a few sample images below. These are images the model had never seen before - not even in the training dataset.

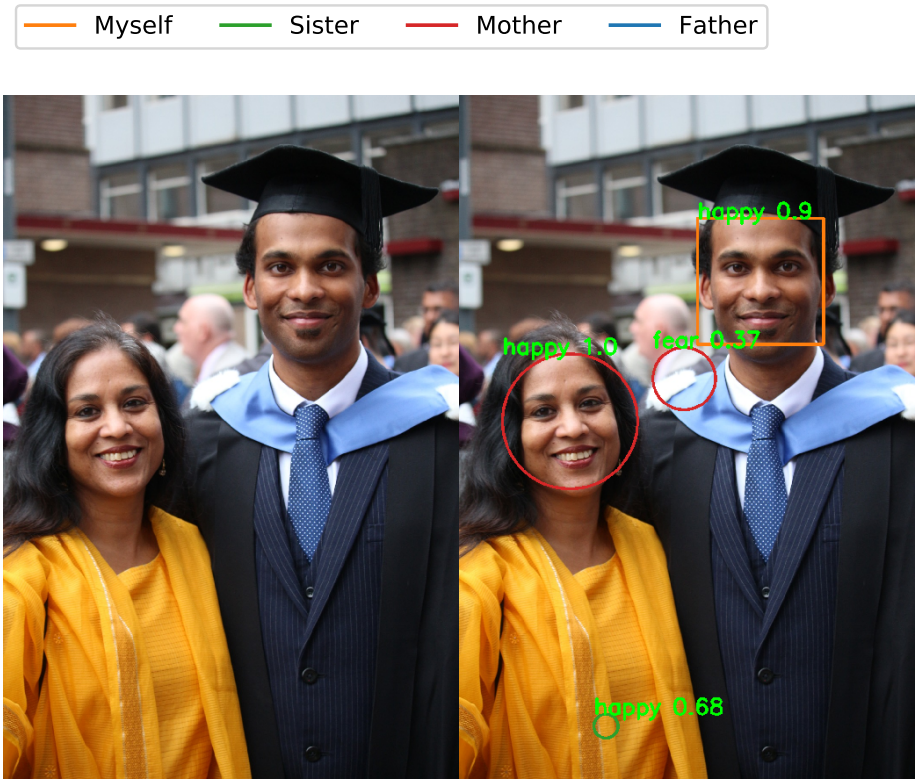

FIGURE 7.1: Inference on sample image.

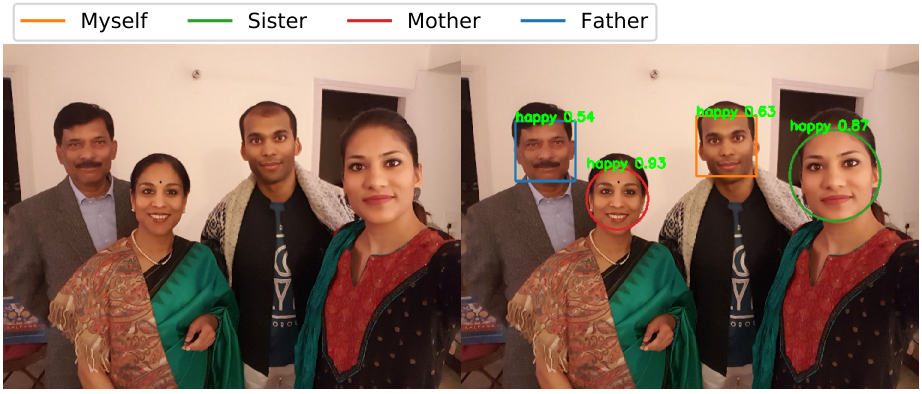

FIGURE 7.2: Model correctly predicts all recognition and gender classes, communicated through the squares for males and circles for females.

We can observe that all three tasks were accomplished seamlessly, and the predictions correspond well to the tangible ground truth (gender classification and facial recognition). The emotions given in the sample images are

homogeneous, but the predicted emotion corresponds well to the emotions present in the pictures.

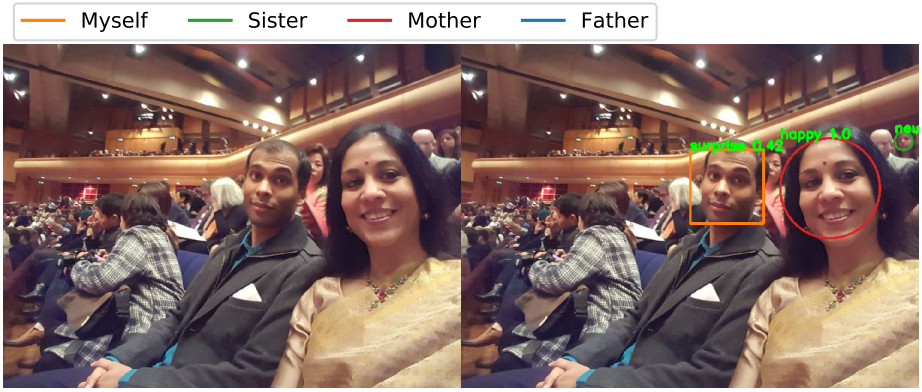

FIGURE 7.3: Inference on sample image.

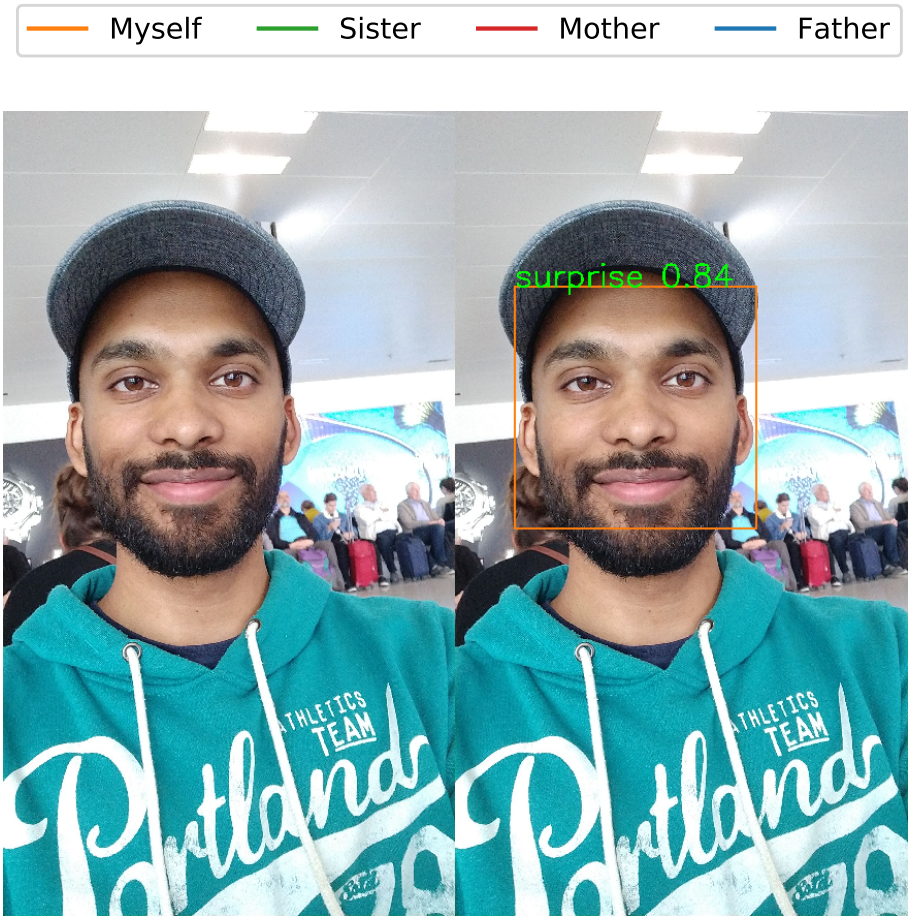

FIGURE 7.4: Facial recognition model correctly predicts class despite beard and occulted hair.

In the next chapter 8, we discuss how the real-time model was implemented, and go over its results.

Chapter 8

Real-Time Model

To challenge and push this project to demonstrate its full potential and maximise our rewards, an attempt was made to implement *another* model which could run our end-to-end model on a *real-time* input feed, such as from a web-cam or a phone camera. This could shift our project from an academic exercise and perspective into the forefront of practical applications.

This prospect had a lot of uncertainty as the actual practicalities of having a deep learning model work on multiple frames per second was completely unknown. But on the upside, the realisation of such a model would be very well-suited for demonstrations. The following sections summarise the process that went into building this script.

8.1 Engineering

In many ways, the implementation of this model was very similar to our 'passive' end-to-end model, as the variables and weights were all loaded and restored in the same way as Listing 7.1, as shown below in Listing 8.1.

```
# Restore all models
reco_values = mm.restore_model(...)
model_fer = recreate_keras_model(...)
gender_values = mm.restore_model(...)
```

Listing 8.1: Restoring the saved models weights.

The web-cam feed could be easy captured with `OpenCV`'s `VideoCapture()` method. It is to be noted that the use of this method often requires administrative permissions, and a direct execution of the script on a new machine may not directly give the expected results. Furthermore, the `VideoCapture()` has a channel feed parameter, which can be 0, 1, or any other integer, depending on the number of output video sources attached to the machine running the script.

The main structure of this script consists of a `while` loop which runs and captures frames until explicitly told to stop. Each frame is then converted from BGR to RGB, and the region of interests, i.e. the faces detected by `OpenCV` are copied onto a new array, which is then pre-processed, before being used for inference of all the tasks.

The pre-processing is the same as the previous models, which includes histogram equalisation, image standardisation and resizing, all of which are accessed through the `MyUtilityMethods` class. The program can be shut at any time by breaking the `while` loop, which can be done by pressing the `Q` key on the user's keyboard. The code given in Listing 8.2 represents the overall skeleton code of the implemented script.

```
frames = []
# Run camera
while(True):
  # Capture frame-by-frame
  ret, frame = cap.read()

  # Our operations on the frame come here
  rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
  frames.append(rgb_frame)

  # Copy image
  image_copy = rgb_frame.copy()

  # Get heads, plot rectangles
  ...

  # Inference
  probs_reco, y_hat_reco = mm.run_model(...)
  probs_fer, y_hat_fer = mm.run_keras_model(...)
  probs_gender, y_hat_gender = mm.run_model(...)

  # Mark-up
  cv2.putText(...)

  # Display the resulting frame
  cv2.imshow('frame', frame)

  # Release keys
  if cv2.waitKey(1) & 0xFF == ord('q'):
    break

# When everything is done, release the capture
cap.release()
cv2.destroyAllWindows()
```

Listing 8.2: Overall structure of script.

8.2 Results

The result was surprisingly robust and legitimate for a prototype implementation. There is indeed little lag, and the model is successfully able to deliver real-time inference of given faces without any major issues. There are other factors which influence the quality of the predictions, such as the lighting conditions, pose, occlusion etc., but these are no different than those in static images. The model is also able to flawlessly perform continuous inference on multiple faces simultaneously.

The results can be seen in the images given below in Figure 8.1, which are best observed by directly running the script, following the instructions given in the user manual in Appendix E.

Alternatively, a sample demonstration video can be viewed at `https://youtu.be/y4KUKlOh9hs`. The legend is identical to the end-to-end model, given through the shapes, colours, and annotated text around the area circumscribing the detected faces.

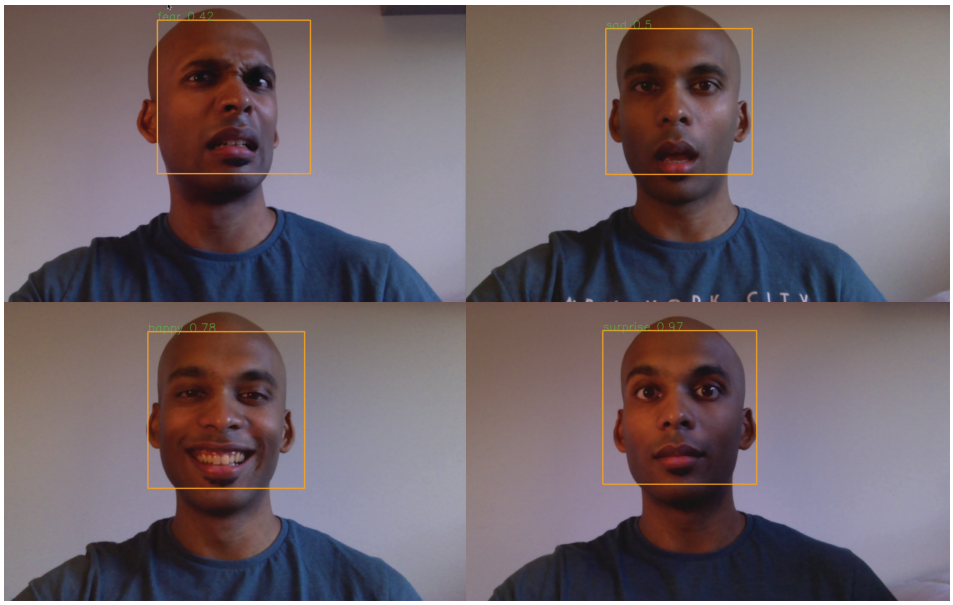

FIGURE 8.1: Sample screen-shots taken when running the real-time model. The predicted emotions and their corresponding probabilities are given in green on the top-left corner of each square. These include the fear, sad, happy, and surprise classes.

In the next chapter 9, we look into critically evaluating the results of our entire project in relation to the goals stated at the start of this thesis, as well as how the models could be further improved in the future.

Chapter 9

Evaluation

To critically assess this project as a whole, one must evaluate it against the goals set out in the project proposal. This chapter attempts to provide an objective evaluation of the project outcomes by comparing the results achieved in relation to the stated goals. Possible future pathways for the project, and the overall learning points are also described subsequently.

9.1 Critical Evaluation

The two lists given in section 1.2 presented the features as planned in the initial proposal and project introduction. We can observe that we have successfully achieved **all** of the planned goals, and in fact gone even further, with the implementation of the real-time model.

Specifically, we have achieved the essential goals with the help of `OpenCV` and our implemented models based on the `MyCNN` and `MyDeepCNN` networks, facilitated by some hyper-parameter tuning.

The desired features were also achieved through the Python utility scripts and with the aid of good design and engineering principles, which allowed for flexibility in all the models. The individual models were also successfully linked into a single pipeline in the end-to-end model, which was able to produce the before-and-after photographs.

9.2 Personal Evaluation

9.2.1 Strengths

The strength of the project lies in its originality, ambition, clarity of vision, experimentations, and technical implementation - all of which are built on a robust theoretical understanding of Deep Learning principles and a comprehensive literature review of contemporary techniques and methodologies. Furthermore, the project work demonstrated how various problems were approached from a scientific perspective, meticulously solved, and then

persistently optimised, based on sound design principles. The project also did a stellar job of systematically building on comprehension, allowing for only incremental increases in complexity, and avoided having to deal with any major bugs.

The project was essentially a multi-dimensional applied data science project, clearly showcasing the strengths and limitations of various deep-learning techniques, and how they can be used to create a substantive product which was potentially capable of improving one's quality of life. It also demonstrated the rigour required to join-up all the various components and make them work together seamlessly. The source code reflects the depth of the experimentation and analysis done for each implemented model, as well as, the many utility methods developed to generate comprehensive metric tools.

This project was also very relevant to current events and context, and perfectly timed as a demonstration of the power of deep learning, and how facial information can be extracted from subjects in a non-invasive way, with or without their consent.

9.2.2 Weaknesses

The weakness in this project is perhaps in its inability to come up with its own novel algorithm or method unique to the problem. Although, all the tasks were solved, and any encountered obstacle was eventually surmounted, it was done through experimentation and analysis based on existing or state-of-the-art techniques, rather than through a new or innovative twist to an existing idea. While the project worked well as a very comprehensive review and application of learnt methods it does not come up with a new or significant technique in itself.

9.3 Future Work

Despite the work done so far, there is still scope for additional experiments to be conducted, as well as, new features to be developed which could further improve the current implementation. The following paragraphs give an insight on such aspects and which could yield even better results for each of the three tasks, and beyond.

The facial recognition task currently works well with model-validation type models. However, presently it is restricted to the classes we include in the training dataset, all of which ideally need to have a minimum quantity of samples to prevent bias in the model. If one were to add a new person to the model, it would mean re-training the entire network, and tuning the

hyper-parameters again. One way to potentially overcome this problem is to implement a *one-shot learning* model, which uses a similarity function to calculate the degree of difference between two given images, and can classify a class based on a **single sample**.

A more exhaustive literature survey and review would be required to fully understand concepts such as *Siamese networks*, which are well explained by Koch, Zemel and Salakhutdinov (2017). The concept is used by Facebook in their exceptional and state-of-the-art application, *DeepFace*, which has an accuracy of **97.35%**, as suggested by Taigman et al. (2014b). Furthermore, learning about *triplet-loss* function would also help to understand other neural networks, such as Google's *FaceNet*, which achieved an accuracy of **99.63%**, as cited in Schroff, Kalenichenko and Philbin (2015).

Emotion analysis and classification is a complex task, and has only achieved a best accuracy of **71%** on the FER2013 dataset. However, one could use the same data augmentation techniques used on our personal data and apply them to the FER2013 dataset to balance and augment it, and see if it results in any improvement in overall accuracy, F1-scores, or Area Under the Curves (AUCs).

Furthermore, it would be very interesting if *new* emotions could be produced, by detecting faces which contain a mix of two different emotions, as shown in Figure 9.1. For example, a softmax model which assigned a sample face the emotions 'fear' and 'disgust', could over a certain minimum threshold consider it as 'outrage'. This idea is based on the concept that emotions are all linked, and complex emotions are expressed as a mix other core emotions. This is similar to colour interactions, where new colours can be created by simply mixing the primary colours in various innumerable proportions and combinations.

Alternatively, one could potentially create a dataset of emotions by using the real-time model to record such emotions by means of a video, and then using individual frames as training input.

Additional ideas to further 'extract' more features from faces could include detecting the ethnicity of a given person, which would again require a labelled dataset.

In relation to the models currently implemented, they could be further tweaked using a grid or random search. Google very recently released a

`Keras Tuner` API which gives users access to such a functionality[1]. Their latest addition includes a *Bayesian* tuner, written specifically for `tf.keras` with TensorFlow `2.0`, currently still a beta version[2].

Finally, one could also implement the real-time model on a website using `tf.js`[3] which would allow one to more easily share the model. One could additionally add toggling functionalities to it, which would allow a user to turn each task, such as gender classification, on or off, in order to play around with the models. This would be significantly more user-friendly than working on the command-line, and could be the perfect way to showcase the work done for this project for a wider audience.

In the next chapter 10, we elaborate on the main learning points of this entire project and the insights gained from it.

[1] Cutting Edge TensorFlow - Keras Tuner: Hyper-tuning for Humans. Google I/O 2019. https://elie.net/talk/cutting-edge-tensorflow-keras-tuner-hypertuning-for-humans. (Accessed on 09/03/2019).

[2] Keras tuner: Hyper-parameter tuning for humans. https://github.com/keras-team/keras-tuner. (Accessed on 09/03/2019)

[3] TensorFlow for JavaScript. https://www.tensorflow.org/js. (Accessed on 09/03/2019).

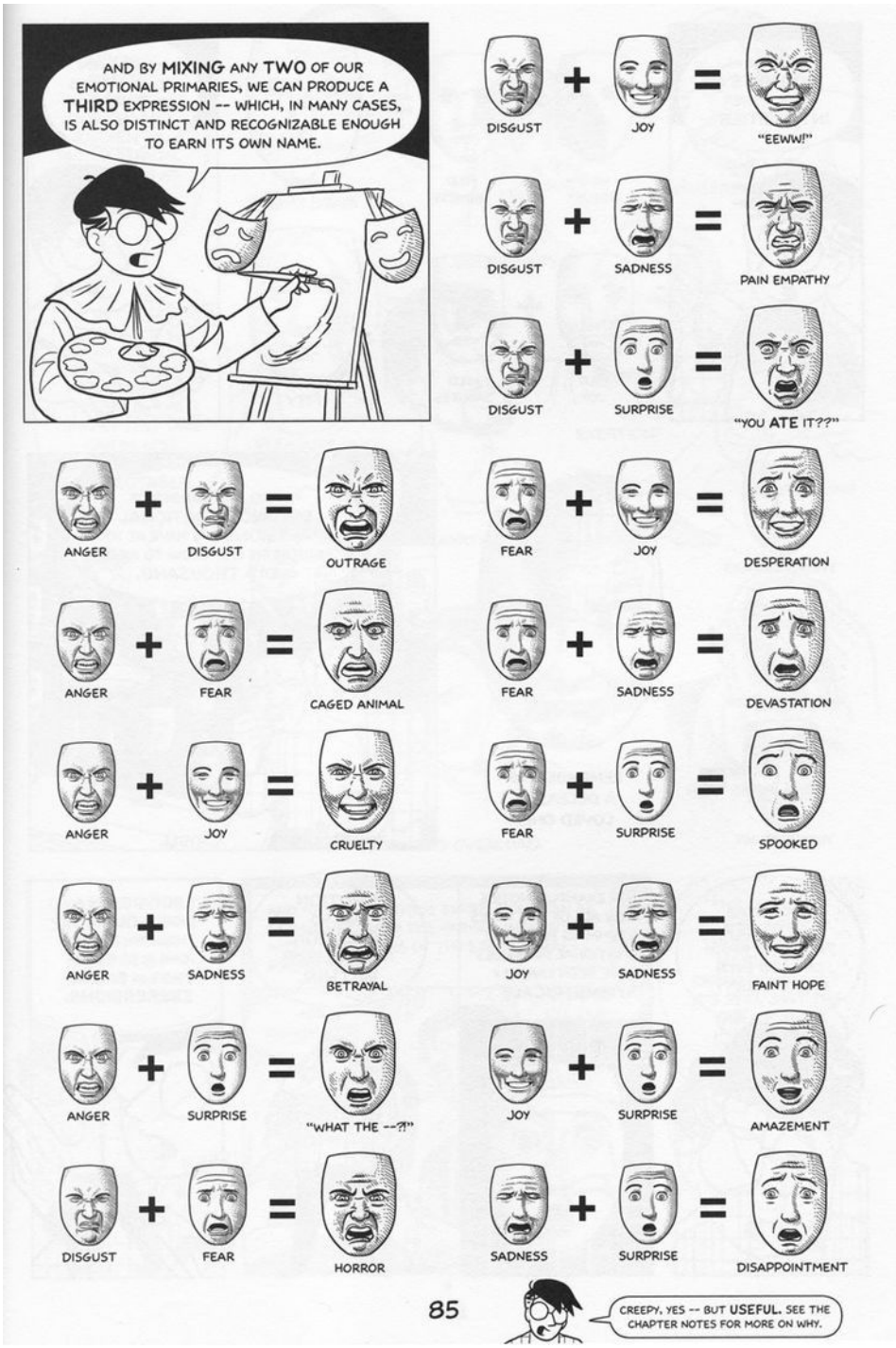

FIGURE 9.1: Mixing emotions to create new ones, from the perspective of an artist. Source: McCloud (2006).

Chapter 10

Learning Points

This project was truly multi-dimensional and technical, which developed and enriched one's skills in several ways. This chapter provides an insight into the main learning points from this project.

From a big picture perspective, the project was first and foremost an education in neural networks, without which one would not have felt truly comfortable with deep learning libraries such as TensorFlow or Keras. Compelled to comprehend abstract concepts and technical nuances, avoid common mistakes, and rigorously understand the data and technologies, one now feels truly empowered to work on new deep learning problems, and further expand one's breadth of technical capacities and potential.

From a more grounded and technical point of view, this project helped realise and appreciate the value and utility of vectorisation, a process which saved an incalculable amount of time at every stage of this large project. Moreover, one gained a more mathematical understanding of how a model's variables and hyper-parameters explicitly interact with convolutional neural networks.

One also truly appreciated the merit of non-technical skills. Indeed, the value of having a clear vision from the very outset became quickly apparent. Furthermore, one realised the critical importance of clearly defining the problem, as well as bounding the project's scope and setting tangible goals. This also positively impacted one's time-management abilities, and highlighted the importance of a strict schedule in order to maintain a productive workflow. Finally, the value of exercising caution, building on comprehension, and increasing complexity only *incrementally*, were perhaps the main factors in the success of this project. Indeed, breaking down the overall problem into three smaller ones, and solving and tweaking each of them was the reason one was able to then put them back together, and in a more complex yet sophisticated way. This resulted in the final end-to-end and real-time models which were the true highlights of this project.

Finally, learning about facial recognition and related technologies also gave one a deeper insight into its power and potentially harmful impact on society. One remarkable outcome of this project is that one is now truly driven to grapple with the cutting-edge of artificial intelligence, especially in the field of computer vision, to the point where one has been engaging with latest developments and publications on a daily basis. One is truly amazed at the potential of artificial intelligence, and ready to apply learnt theoretical knowledge and practical experience to real-world projects, in order to harness its power, and play one's part in positively shaping the AI-dominated world of tomorrow.

Appendix A

Hyper-parameters

A.1 Facial Recognition

Table A.1: Hold-out Model

Hyper-parameter	Value
Network	`MyCNN`
Validation type	Hold-out
Number of epochs	50
Learning Rate	1×10^{-5}
Batch size	30
Final activation function	Softmax
Random seed	0

Table A.2: K-fold Model

Hyper-parameter	Value
Network	`MyCNN`
Validation type	k-fold
Number of epochs	50
Learning Rate	1×10^{-5}
Batch size	30
Final activation function	Softmax
Random seed	0

A.2 Gender Classification

Table A.3: Softmax Model

Hyper-parameter	Value
Network	`MyCNN`
Validation type	k-fold
Number of epochs	50
Learning Rate	1×10^{-5}
Batch size	30
Final activation function	Softmax
Random seed	0

Table A.4: Sigmoid Model

Hyper-parameter	Value
Network	`MyCNN`
Validation type	k-fold
Number of epochs	50
Learning Rate	1×10^{-5}
Batch size	30
Final activation function	Sigmoid
Random seed	0

A.3 Emotion Classification

Table A.5: Hold-out Model

Hyper-parameter	Value
Network	`MyCNN`
Validation type	Hold-out
Number of epochs	50
Learning Rate	1×10^{-6}
Batch size	30
Final activation function	Softmax
Random seed	0

Table A.6: K-Fold Model

Hyper-parameter	Value
Network	`MyCNN`
Validation type	k-fold
Number of epochs	50
Learning Rate	1×10^{-6}
Batch size	30
Final activation function	Softmax
Random seed	0

Table A.7: Deep Network Model

Hyper-parameter	Value
Network	`MyDeepCNN`
Validation type	Hold-out
Number of epochs	50
Learning Rate	1×10^{-6}
Batch size	32
Final activation function	Softmax
Random seed	0
Beta 1	1×10^{-6}
Beta 2	1×10^{-6}
Epsilon	1×10^{-6}
Number of features	32

Appendix B

ROC Curves

B.1 Facial Recognition

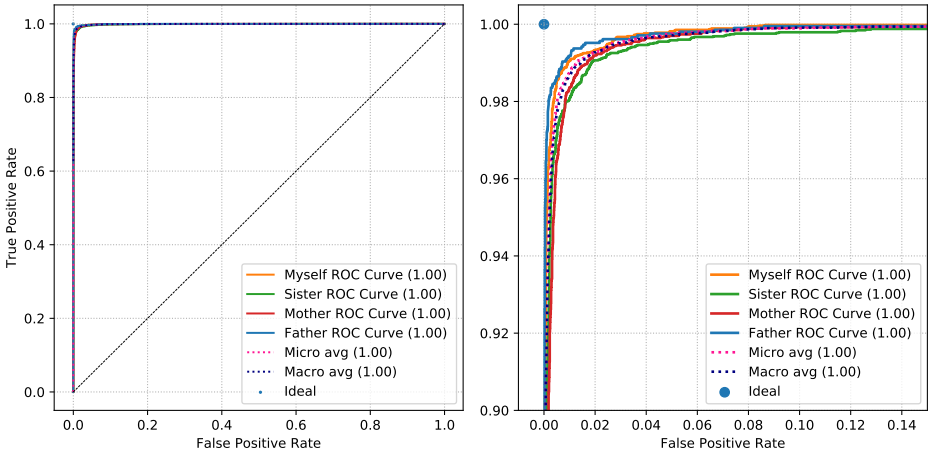

FIGURE B.1: Hold-out model training ROC curves.

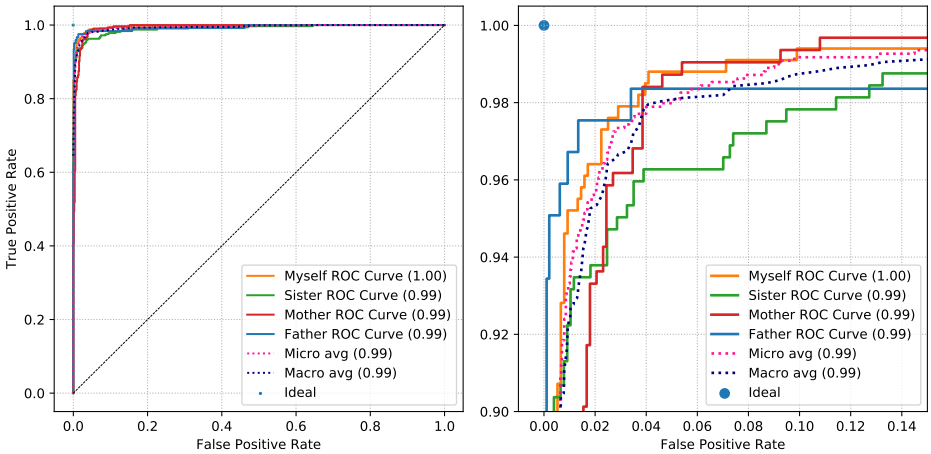

FIGURE B.2: Hold-out model testing ROC curves.

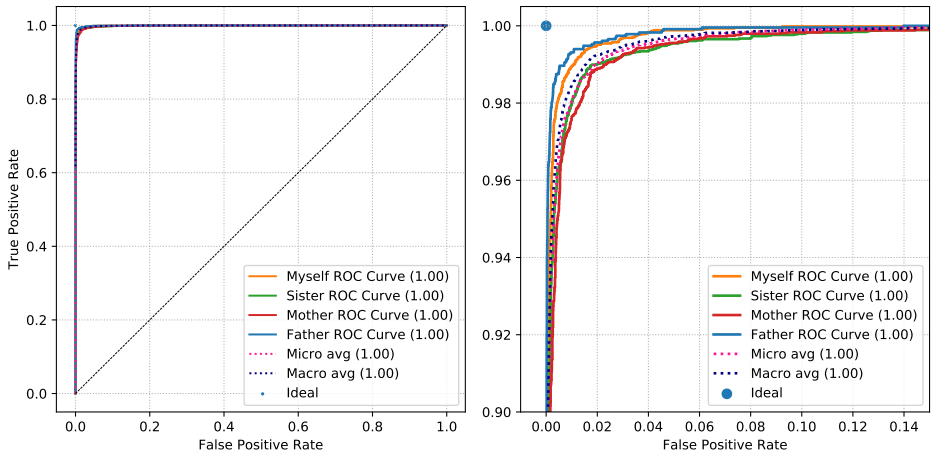

FIGURE B.3: K-fold model training ROC curves.

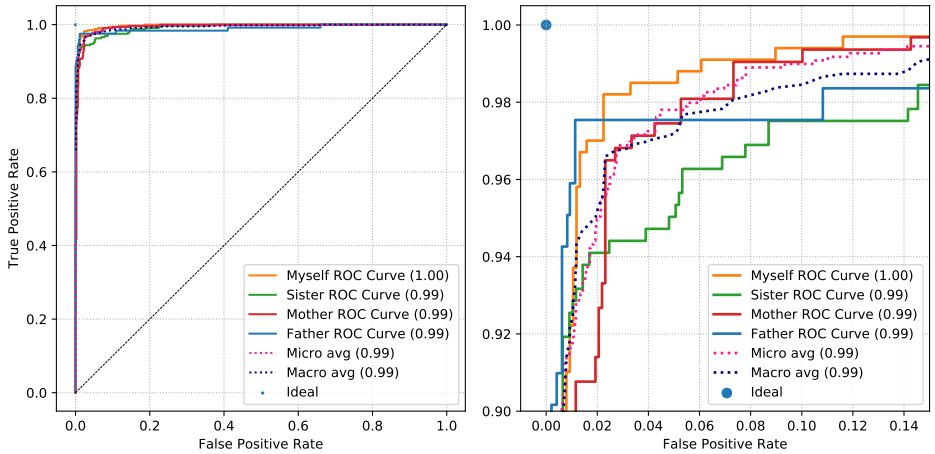

FIGURE B.4: K-fold model training ROC curves.

B.2 Gender Classification

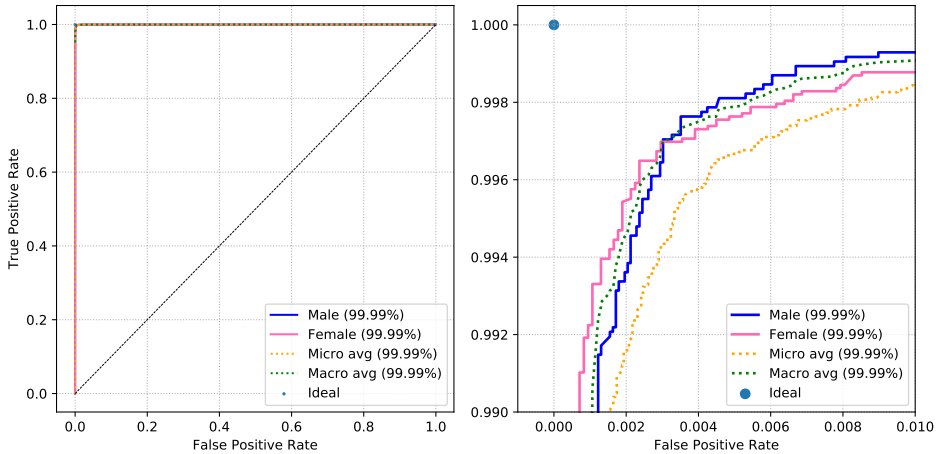

FIGURE B.5: Softmax model training ROC curves.

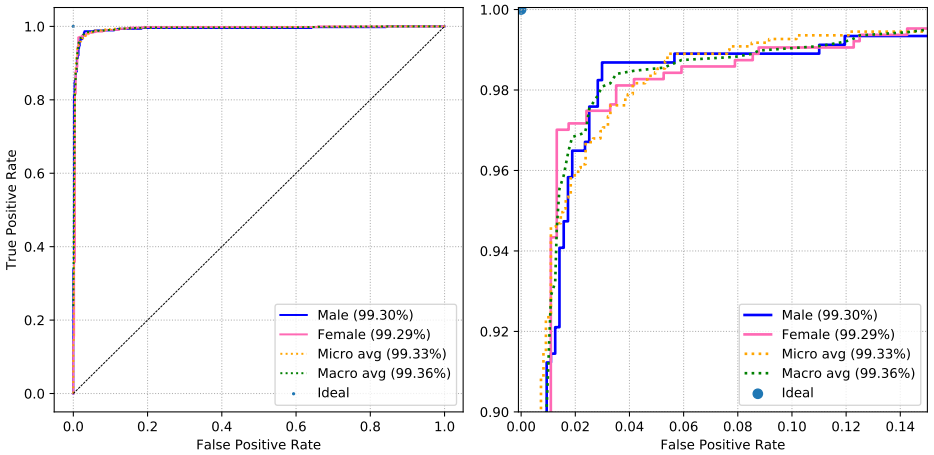

FIGURE B.6: Softmax model testing ROC curves.

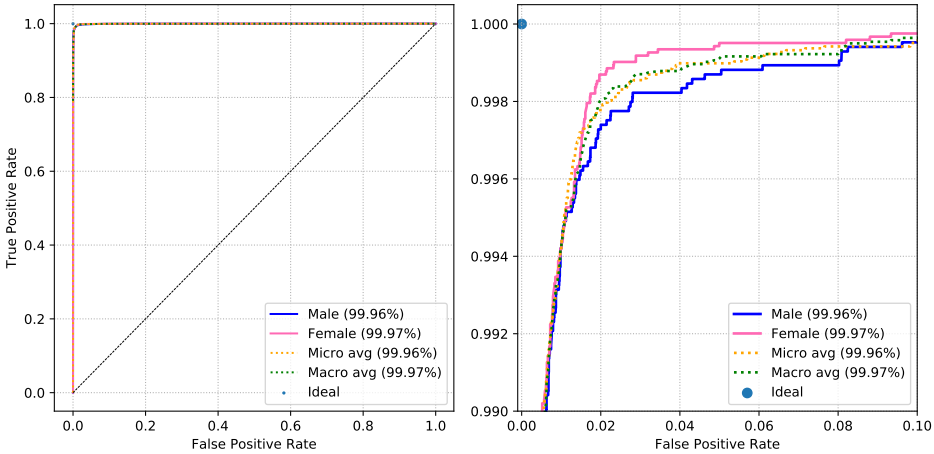

FIGURE B.7: Sigmoid model training ROC curves.

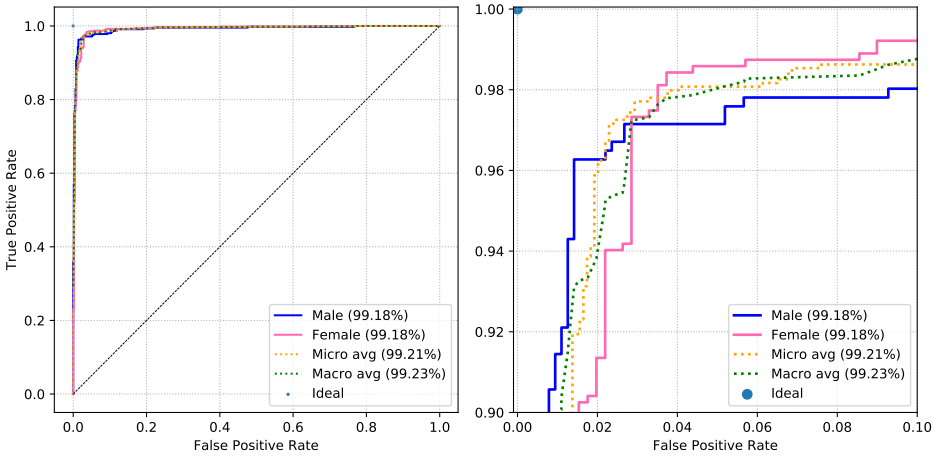

FIGURE B.8: Sigmoid model testing ROC curves.

B.3 Emotion Detection

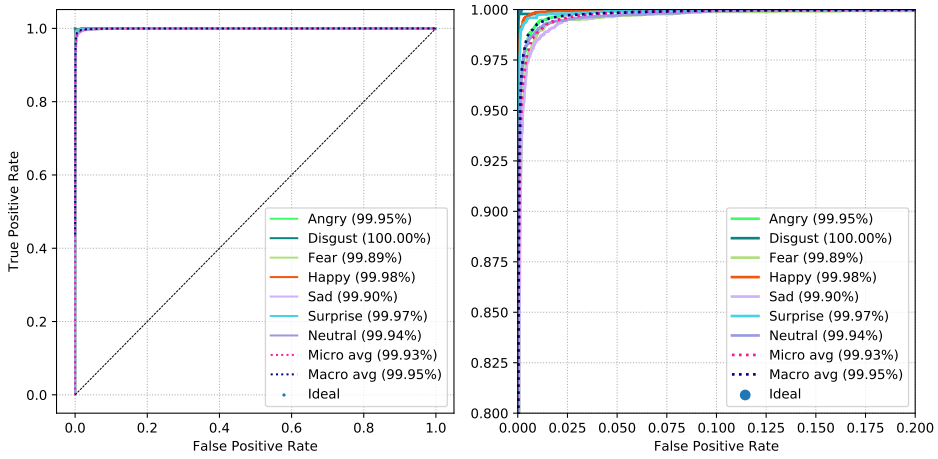

FIGURE B.9: Hold-out model training ROC curves.

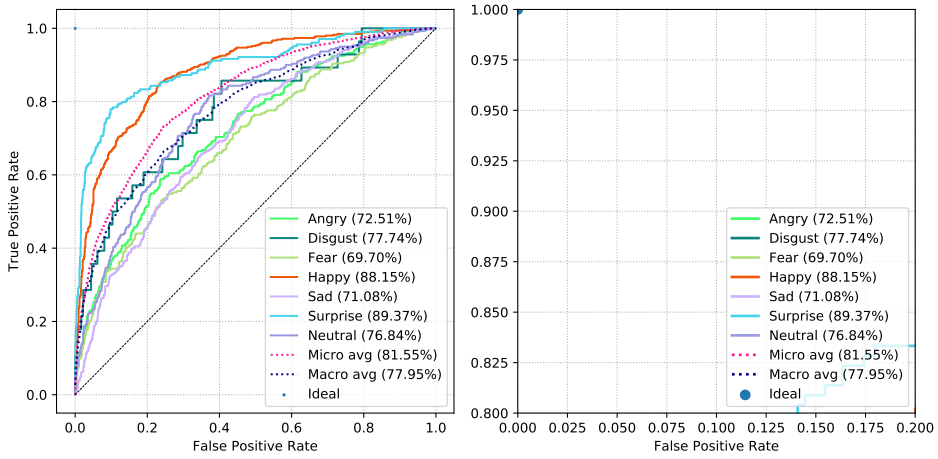

FIGURE B.10: Hold-out model testing ROC curves.

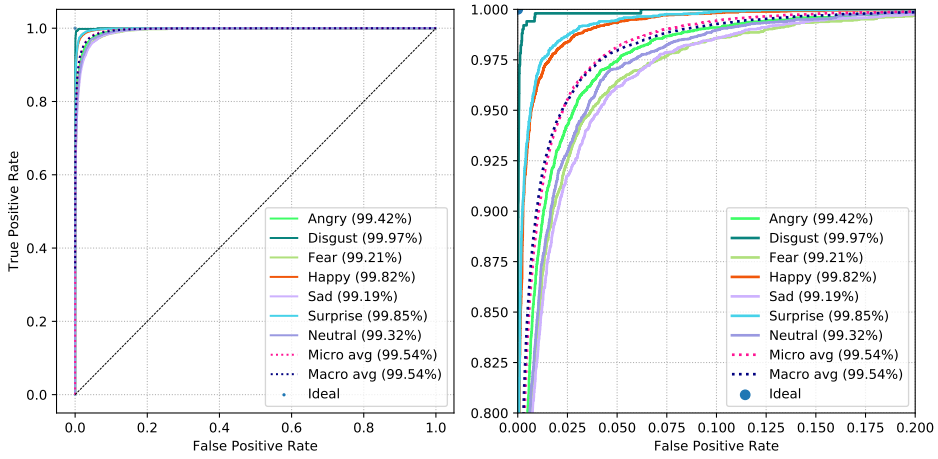

FIGURE B.11: K-fold model training ROC curves.

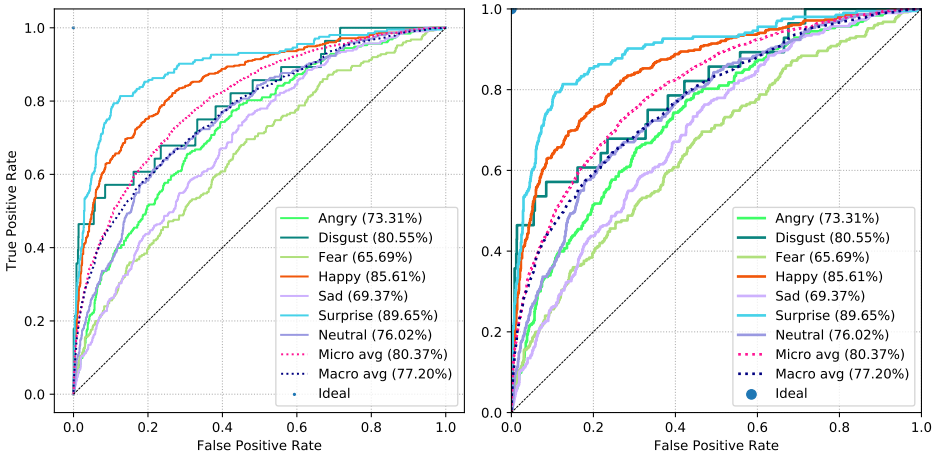

FIGURE B.12: K-fold model testing ROC curves.

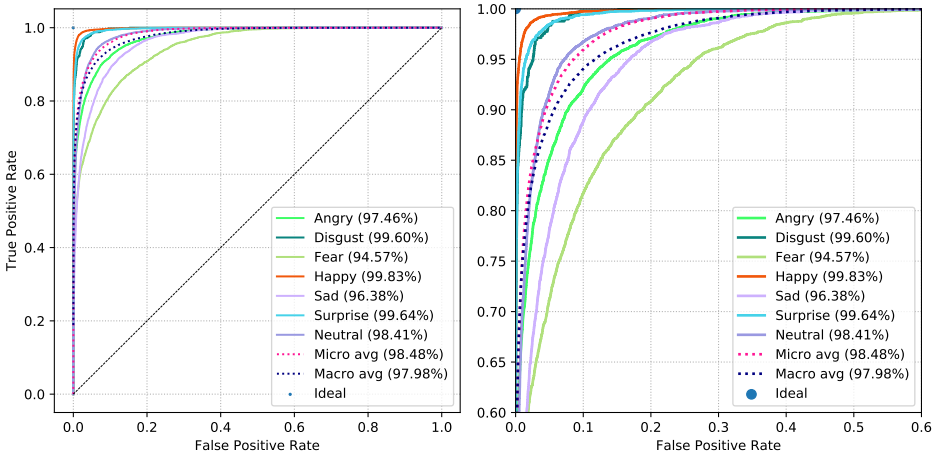

FIGURE B.13: Deep model training ROC curves.

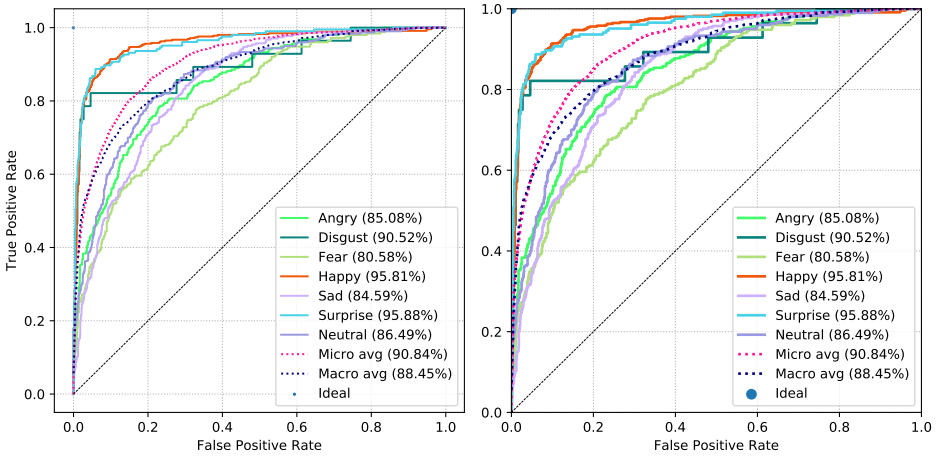

FIGURE B.14: Deep model testing ROC curves.

Appendix C

Metric Tables

C.1 Facial Recognition

C.1.1 Hold-out Model

Table C.1: Training metrics per class.

	f1-score	precision	recall	support
Myself	0.99	0.99	0.99	5469.00
Sister	0.98	0.98	0.98	5772.00
Mother	0.98	0.98	0.98	5318.00
Father	0.98	0.99	0.98	2071.00
accuracy	0.98	0.98	0.98	0.98
macro avg	0.98	0.98	0.98	18630.00
weighted avg	0.98	0.98	0.98	18630.00

Table C.2: Validation metrics per class.

	f1-score	precision	recall	support
Myself	0.94	0.95	0.92	659.00
Sister	0.93	0.92	0.93	600.00
Mother	0.93	0.92	0.94	560.00
Father	0.97	0.96	0.97	251.00
accuracy	0.94	0.94	0.94	0.94
macro avg	0.94	0.94	0.94	2070.00
weighted avg	0.94	0.94	0.94	2070.00

Table C.3: Testing metrics per class.

	f1-score	precision	recall	support
Myself	0.96	0.95	0.96	334.00
Sister	0.95	0.96	0.94	322.00
Mother	0.94	0.94	0.95	314.00
Father	0.95	0.96	0.95	122.00
accuracy	0.95	0.95	0.95	0.95
macro avg	0.95	0.95	0.95	1092.00
weighted avg	0.95	0.95	0.95	1092.00

C.1.2 K-fold Model

Table C.4: Training metrics per class.

	f1-score	precision	recall	support
Myself	0.98	0.99	0.97	6128.00
Sister	0.97	0.96	0.99	6372.00
Mother	0.97	0.97	0.98	5878.00
Father	0.98	0.99	0.97	2322.00
accuracy	0.98	0.98	0.98	0.98
macro avg	0.98	0.98	0.97	20700.00
weighted avg	0.98	0.98	0.98	20700.00

Table C.5: Validation metrics per class.

	f1-score	precision	recall	support
Myself	0.97	0.99	0.95	138.00
Sister	0.96	0.93	0.99	111.00
Mother	0.97	0.97	0.97	120.00
Father	0.99	1.00	0.98	45.00
accuracy	0.97	0.97	0.97	0.97
macro avg	0.97	0.97	0.97	414.00
weighted avg	0.97	0.97	0.97	414.00

Table C.6: Testing metrics per class.

	f1-score	precision	recall	support
Myself	0.97	0.96	0.97	334.00
Sister	0.93	0.92	0.94	322.00
Mother	0.94	0.94	0.94	314.00
Father	0.93	0.96	0.90	122.00
accuracy	0.95	0.95	0.95	0.95
macro avg	0.94	0.95	0.94	1092.00
weighted avg	0.95	0.95	0.95	1092.00

C.2 Gender Classification

C.2.1 Softmax Model

Table C.7: Training metrics per class.

	precision	recall	f1-score	support
Male	1.0	0.99	1.0	8450.0
Female	1.0	1.00	1.0	12250.0
accuracy	1.0	1.00	1.0	1.0
macro avg	1.0	1.00	1.0	20700.0
weighted avg	1.0	1.00	1.0	20700.0

Table C.8: Validation metrics per class.

	precision	recall	f1-score	support
Male	0.99	0.99	0.99	183.0
Female	1.00	1.00	1.00	231.0
accuracy	1.00	1.00	1.00	1.0
macro avg	1.00	1.00	1.00	414.0
weighted avg	1.00	1.00	1.00	414.0

Table C.9: Testing metrics per class.

	precision	recall	f1-score	support
Male	0.98	0.96	0.97	456.00
Female	0.97	0.98	0.98	636.00
accuracy	0.98	0.98	0.98	0.98
macro avg	0.98	0.97	0.97	1092.00
weighted avg	0.98	0.98	0.98	1092.00

C.2.2 Sigmoid Model

TABLE C.10: Training metrics per class.

	precision	recall	f1-score	support
Male	0.99	0.99	0.99	8450.00
Female	0.99	0.99	0.99	12250.00
accuracy	0.99	0.99	0.99	0.99
macro avg	0.99	0.99	0.99	20700.00
weighted avg	0.99	0.99	0.99	20700.00

TABLE C.11: Validation metrics per class.

	precision	recall	f1-score	support
Male	0.98	0.97	0.98	183.00
Female	0.98	0.99	0.98	231.00
accuracy	0.98	0.98	0.98	0.98
macro avg	0.98	0.98	0.98	414.00
weighted avg	0.98	0.98	0.98	414.00

TABLE C.12: Testing metrics per class.

	precision	recall	f1-score	support
Male	0.97	0.96	0.97	456.00
Female	0.97	0.98	0.98	636.00
accuracy	0.97	0.97	0.97	0.97
macro avg	0.97	0.97	0.97	1092.00
weighted avg	0.97	0.97	0.97	1092.00

C.3 Emotion Detection

C.3.1 Hold-out Model

Table C.13: Training metrics per class.

	f1-score	precision	recall	support
Angry	0.96	0.99	0.93	4226.00
Disgust	0.99	0.99	0.98	457.00
Fear	0.96	0.92	0.99	4373.00
Happy	0.99	0.99	0.99	7677.00
Sad	0.97	0.98	0.96	5212.00
Surprise	0.98	1.00	0.96	3426.00
Neutral	0.98	0.97	0.99	5301.00
accuracy	0.97	0.97	0.97	0.97
macro avg	0.97	0.98	0.97	30672.00
weighted avg	0.97	0.98	0.97	30672.00

Table C.14: Validation metrics per class.

	f1-score	precision	recall	support
Angry	0.35	0.39	0.32	466.00
Disgust	0.47	0.56	0.40	62.00
Fear	0.36	0.32	0.41	498.00
Happy	0.65	0.64	0.65	856.00
Sad	0.34	0.36	0.33	560.00
Surprise	0.61	0.66	0.56	371.00
Neutral	0.42	0.40	0.43	596.00
accuracy	0.47	0.47	0.47	0.47
macro avg	0.46	0.48	0.44	3409.00
weighted avg	0.47	0.47	0.47	3409.00

Table C.15: Testing metrics per class.

	f1-score	precision	recall	support
Angry	0.36	0.40	0.32	253.00
Disgust	0.19	0.27	0.14	28.00
Fear	0.35	0.31	0.39	250.00
Happy	0.68	0.69	0.66	455.00
Sad	0.35	0.36	0.35	304.00
Surprise	0.66	0.73	0.61	204.00
Neutral	0.43	0.39	0.47	300.00
accuracy	0.48	0.48	0.48	0.48
macro avg	0.43	0.45	0.42	1794.00
weighted avg	0.48	0.49	0.48	1794.00

C.3.2 K-fold Model

Table C.16: Training metrics per class.

	f1-score	precision	recall	support
Angry	0.90	0.93	0.88	4561.00
Disgust	0.95	0.94	0.97	506.00
Fear	0.90	0.90	0.89	4728.00
Happy	0.96	0.94	0.98	8311.00
Sad	0.89	0.84	0.94	5600.00
Surprise	0.95	0.96	0.94	3695.00
Neutral	0.89	0.95	0.85	5749.00
accuracy	0.92	0.92	0.92	0.92
macro avg	0.92	0.92	0.92	33150.00
weighted avg	0.92	0.92	0.92	33150.00

TABLE C.17: Validation metrics per class.

	f1-score	precision	recall	support
Angry	0.86	0.88	0.84	89.00
Disgust	1.00	1.00	1.00	11.00
Fear	0.85	0.89	0.82	89.00
Happy	0.93	0.89	0.96	169.00
Sad	0.85	0.79	0.93	109.00
Surprise	0.92	0.94	0.90	73.00
Neutral	0.81	0.88	0.75	123.00
accuracy	0.88	0.88	0.88	0.88
macro avg	0.89	0.90	0.89	663.00
weighted avg	0.88	0.88	0.88	663.00

TABLE C.18: Testing metrics per class.

	f1-score	precision	recall	support
Angry	0.37	0.39	0.35	253.00
Disgust	0.36	0.37	0.36	28.00
Fear	0.27	0.28	0.27	250.00
Happy	0.65	0.64	0.66	455.00
Sad	0.34	0.32	0.37	304.00
Surprise	0.59	0.57	0.62	204.00
Neutral	0.38	0.42	0.35	300.00
accuracy	0.45	0.45	0.45	0.45
macro avg	0.42	0.43	0.43	1794.00
weighted avg	0.45	0.45	0.45	1794.00

C.3.3 Deep Model

Table C.19: Training metrics per class.

	f1-score	precision	recall	support
Angry	0.78	0.72	0.85	4441.00
Disgust	0.79	0.74	0.85	489.00
Fear	0.70	0.84	0.61	4608.00
Happy	0.97	0.96	0.97	8095.00
Sad	0.77	0.79	0.76	5458.00
Surprise	0.93	0.95	0.90	3599.00
Neutral	0.85	0.81	0.91	5597.00
accuracy	0.85	0.85	0.85	0.85
macro avg	0.83	0.83	0.84	32287.00
weighted avg	0.84	0.85	0.85	32287.00

Table C.20: Validation metrics per class.

	f1-score	precision	recall	support
Angry	0.56	0.48	0.67	251.00
Disgust	0.54	0.55	0.53	30.00
Fear	0.42	0.53	0.35	263.00
Happy	0.84	0.83	0.86	438.00
Sad	0.49	0.53	0.46	314.00
Surprise	0.77	0.83	0.71	198.00
Neutral	0.59	0.55	0.63	300.00
accuracy	0.63	0.63	0.63	0.63
macro avg	0.60	0.61	0.60	1794.00
weighted avg	0.62	0.63	0.63	1794.00

TABLE C.21: Testing metrics per class.

	f1-score	precision	recall	support
Angry	0.51	0.45	0.58	253.00
Disgust	0.45	0.41	0.50	28.00
Fear	0.41	0.54	0.34	250.00
Happy	0.84	0.82	0.86	455.00
Sad	0.50	0.54	0.46	304.00
Surprise	0.78	0.82	0.74	204.00
Neutral	0.59	0.54	0.65	300.00
accuracy	0.62	0.62	0.62	0.62
macro avg	0.58	0.59	0.59	1794.00
weighted avg	0.62	0.63	0.62	1794.00

Appendix D

Source Codes

The sections below give the souce codes for all implemented classes, methods, and utility scripts. These include the following files:

- `myutilitymethods.py`
- `mycnn.py`
- `mydeepcnn.py`
- `RestoreModel.py`
- `LiveInput.py`
- `prepare_cropped_faces.py`
- `convert_images_to_data.py`

D.1 My Utility Methods

```python
import os
import sys
import cv2
import time
import keras
import random
import logging
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from scipy import interp
import albumentations as A
from itertools import cycle
from IPython import display
from sklearn import metrics
from datetime import datetime
#from mtcnn.mtcnn import MTCNN
import matplotlib.colors as mc
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.decomposition import PCA
from matplotlib.patches import Rectangle
from sklearn.metrics import classification_report
from sklearn.utils.multiclass import unique_labels
from sklearn.metrics import confusion_matrix as sk_cm

class MyMethods:

  def __init__(self, NUM_CLASSES, num_to_class, class_to_num, class_colors=['C1', 'C2', 'C3', 'C0']):
    '''Initialise with NUM_CLASSES, num_to_class, clams_to_num'''
    self.NUM_CLASSES = NUM_CLASSES
    self.num_to_class = num_to_class
    self.class_to_num = class_to_num
    if self.NUM_CLASSES != 4:
      class_colors = self.randomColorGenerator(self.NUM_CLASSES)
    self.class_colors = class_colors

  def read_csv(filepath, aug_factor=10, print_progress=False, img_size=28):
    '''Takes in .csv and returns the compressed images's pixels in RGB format as Numpy array'''
    img = pd.read_csv(filepath, header=None).values
    return img.reshape(img.shape[0], img_size, img_size, 3)

  def equalise_image(self, image, eq_type='HSV'):
    '''Return HSV or YCrCb equalised image'''
    if eq_type == 'HSV':
      H, S, V = cv2.split(cv2.cvtColor(image, cv2.COLOR_RGB2HSV))
      eq_V = cv2.equalizeHist(V)
      return cv2.cvtColor(cv2.merge([H, S, eq_V]), cv2.COLOR_HSV2RGB)
    elif eq_type == 'YCrCb':
      Y, Cr, Cb = cv2.split(cv2.cvtColor(image, cv2.COLOR_RGB2YCrCb))
      eq_Y = cv2.equalizeHist(Y)
      return cv2.cvtColor(cv2.merge([eq_Y, Cr, Cb]), cv2.COLOR_YCrCb2RGB)

  def equalise_gray_image(self, image, eq_type='HSV'):
    '''Return HSV or YCrCb equalised image'''
    if eq_type == 'HSV':
      H, S, V = cv2.split(cv2.cvtColor(image, cv2.COLOR_GRAY))
      eq_V = cv2.equalizeHist(V)
      return cv2.cvtColor(cv2.merge([H, S, eq_V]), cv2.COLOR_HSV2RGB)
    elif eq_type == 'YCrCb':
      Y, Cr, Cb = cv2.split(cv2.cvtColor(image, cv2.COLOR_RGB2YCrCb))
      eq_Y = cv2.equalizeHist(Y)
      return cv2.cvtColor(cv2.merge([eq_Y, Cr, Cb]), cv2.COLOR_YCrCb2RGB)

  def convertToRGB(self, image):
    '''Convert image from BGR to RGB'''
    return cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

  def resize_image(self, img, img_size=28):
    '''Resize image to chosen size'''
    return cv2.resize(img, (img_size, img_size))

  def standardise_images(self, images):
    '''Standardise a batch of images'''
    std_images = []
    for img in images:
      std_images.append(self.standardise_image(img))
    return std_images

  def standardise_image(self, image):
    '''Standardise an image per channel'''
    means = image.mean(axis=(0,1))
    stds = image.std(axis=(0,1))
    return (image - means) / stds

  def normalise_image(self, image, norm_range=(0,1)):
    '''Normalise an image per channel'''
    left_term = norm_range[1]-norm_range[0]
    middle_term = (image-image.min(axis=(0,1))) / (image.max(axis=(0,1)) - image.min(axis=(  0,  1)))
    right_term = norm_range[0]
    return left_term * middle_term + right_term

  def greyscale_image(self, image):
    '''Greyscale an RGB image'''
    return cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

  def print_stats(self, mypic):
    '''Print min, max, mean, std, and variance of a given image'''
    print('Min :', np.round(mypic.min(axis=(0,1)),  2))
    print('Max :', np.round(mypic.max(axis=(0,1)),  2))
    print('Mean:', np.round(mypic.mean(axis=(0,1)), 2))
    print('Std :', np.round(mypic.std(axis=(0,1)),  2))
    print('Var :', np.round(mypic.var(axis=(0,1)),  2))

  def describe_data(self, x_train, y_train, x_test, y_test):
    '''Describe train and test data'''
    print('Shapes')
    print('x_train :', x_train.shape)
    print('y_train :', y_train.shape)
    print('x_test  :', x_test.shape)
    print('y_test  :', y_test.shape)
    print('')
    print('Min')
    print('x_train :', np.round(x_train.min(),3))
    print('x_test  :', np.round(x_test.min(), 3))
    print('')
    print('Max')
    print('x_train :', np.round(x_train.max(),3))
    print('x_test  :', np.round(x_test.max(), 3))
    print('')
    print('Means')
    print('x_train :', np.round(x_train.mean(),3))
    print('x_test  :', np.round(x_test.mean(), 3))
    print('')
    print('Std')
    print('x_train :', np.round(x_train.std(),3))
    print('x_test  :', np.round(x_test.std(), 3))


  def one_hot_encode(self, y):
    '''One-hot encode 1D list of labels'''
    return np.eye(self.NUM_CLASSES)[y]

  def split_train_test(self, x, y, split=0.9):
    '''Split dataset into train and test sets'''
    x_train = x[:int(len(x)*split)]
    y_train = y[:int(len(y)*split)]
    x_test = x[int(len(x)*split):]
    y_test = y[int(len(y)*split):]
    return x_train, y_train, x_test, y_test

  def split_train_val_test(self, x, y, split1=0.9, split2=0.95):
    '''Split dataset into train, validation, and test sets'''
    x_train, x_val, x_test = np.split(x, [int(len(x)*split1), int(len(x)*split2)])
    y_train, y_val, y_test = np.split(y, [int(len(y)*split1), int(len(y)*split2)])
    return x_train, y_train, x_val, y_val, x_test, y_test

  def compute_rolling_average(self, myList):
    '''Computes and returns the rolling average of a given list'''
    rolling_avg = []
    for i, v in enumerate(myList, 1):
      rolling_avg.append(np.sum(myList[:i])/i)
    return rolling_avg

  def plot_results(self, x_test, y_real, y_hat, probs, sleep_time=3.0, dynamic=True, include_title=True):
    '''Plot image and probability distribution bar plot of faces'''
    for i, pic in enumerate(x_test):
      fig, (ax1,ax2) = plt.subplots(ncols=2, figsize=(5,2.5), constrained_layout=True, dpi=  100)
      if include_title:
        fig.suptitle(f'Actual:{self.num_to_class[y_real[i]]}, Predicted:{self.num_to_class[y_hat[i]  ]}', fontsize=15)
      # Left
      ax1.imshow(cv2.resize(x_test[i], (x_test[i].shape[0], x_test[i].shape[0])))
      ax1.axes.get_xaxis().set_visible(False)
      ax1.axes.get_yaxis().set_visible(False)
      # Right
      ax2.barh(np.arange(self.NUM_CLASSES), probs[i])
      ax2.set_yticks(np.arange(self.NUM_CLASSES))
      ax2.set_yticklabels(self.num_to_class)
      ax2.invert_yaxis()
      ax2.set_xlim(left=0.0, right=1)
      # Method 1
      # Plot
      #plt.show()
      # GIF
      #time.sleep(sleep_time)
      #display.clear_output(wait=True)
      # Method 2
      if dynamic:
        display.clear_output(wait=True)
        display.display(plt.gcf())
        time.sleep(sleep_time)
      else:
        plt.show()

  def plot_metrics(self, accuracy, loss, val_accuracy=None, val_loss=None, dpi=100):
    '''Plot graph subplots of the accuracy and loss of the model over the epochs during training.'''
    fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(8,5), constrained_layout=True, dpi=  dpi)
    fig.suptitle('Metrics', fontsize=15)
    # Plot
    axes[0,0].plot(accuracy, label='Train Accuracy')
    axes[1,0].plot(loss, label='Train Loss')
    axes[0,1].plot(self.compute_rolling_average(accuracy), label='Average Train Accuracy')
    axes[1,1].plot(self.compute_rolling_average(loss), label='Average Train Loss')
    if val_accuracy is not None and val_loss is not None:
      axes[0,0].plot(val_accuracy, label='Validation Accuracy')
      axes[1,0].plot(val_loss, label='Validation Loss')
      axes[0,1].plot(self.compute_rolling_average(val_accuracy), label='Average Validation   Accuracy')
      axes[1,1].plot(self.compute_rolling_average(val_loss), label='Average Validation Loss')
    # Titles
    axes[0,0].set_title('Accuracy')
    axes[1,0].set_title('Loss')
    axes[0,1].set_title('Rolling Accuracy Average')
    axes[1,1].set_title('Rolling Loss Average')
    # Axis
    #axes[0,0].set_xlabel('Iterations')
    #axes[0,0].set_ylabel('Accuracy')
    #axes[1,0].set_xlabel('Loss')
    #axes[0,1].set_title('Rolling Accuracy Average')
    #axes[1,1].set_title('Rolling Loss Average')
    # Limits
    axes[0,0].set_ylim(bottom=0.0, top=1)
    axes[0,1].set_ylim(bottom=0.0, top=1)
    # Plot
    if val_accuracy is not None and val_loss is not None:
      plt.legend()
    plt.show()


  def plot_pca(self, x_train, y_train, dpi=150, title=None):
    '''Plot PCA plot of training set'''
```

D.2 My Utility Methods

```
    n_train = x_train.shape[0]
    nb_features = np.prod(x_train.shape[1:])
    x_train = x_train.reshape((n_train, nb_features))
    # PCA
    pca = PCA(n_components=2)
    x_fit = pca.fit_transform(x_train)
    # Plot
    plt.figure(figsize=(8,5), dpi=dpi)
    for num in set(y_train):
      #plt.scatter(x_fit[y_train==num, 0], x_fit[y_train==num, 1], label=num_to_class[num],   color=class_colors[num], s=1)
      plt.scatter(x_fit[y_train==num, 0], x_fit[y_train==num, 1], s=1)
    if title:
      plt.title(title)
    #plt.legend(loc=2)
    plt.show()

def get_false_classifications(self, y_real, all_preds):
    '''Get false classification values'''
    return np.where((y_real == all_preds)*1 == 0)[0]

def plot_confusion_matrix(self, y_true, y_pred, classes, normalize=False, title=None, cmap='GnBu',   dpi=100):
    '''Plot confusion matrix with y_real, y_hat, and classes.'''
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'
    # Compute confusion matrix
    cm = sk_cm(y_true, y_pred)
    # Only use the labels that appear in the data
    classes = classes[unique_labels(y_true, y_pred)]
    if normalize: cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    # Subplots
    fig, ax = plt.subplots(dpi=dpi)
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # Label
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           xticklabels=classes,
           yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')
    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_yticklabels(), rotation=90, ha="right", rotation_mode="anchor")
    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    # Loop
    for i in range(cm.shape[0]):
      for j in range(cm.shape[1]):
        ax.text(j, i, format(cm[i, j], fmt),
                ha="center", va="center",
                color="white" if cm[i, j] > thresh else "black")
    # Plot
    fig.tight_layout()
    plt.show()

def plot_all_confusion_matrices(y_true, y_pred, y_true_val, y_pred_val, y_true_test, y_pred_test, classes, normalize=False, title=None, cmap='GnBu',
    '''Plot train, validation, and test confusion matrices'''
    if not title:
      if normalize:
        title = 'Normalized Confusion Matrices'
      else:
        title = 'Non-Normalized Confusion Matrices'
    # Compute confusion matrix
    cm_train = sk_cm(y_true, y_pred)
    cm_val = sk_cm(y_true_val, y_pred_val)
    cm_test = sk_cm(y_true_test, y_pred_test)
    # Only use the labels that appear in the data
    classes = classes[unique_labels(y_true, y_pred)]
    if normalize:
      cm_train = cm_train.astype('float') / cm_train.sum(axis=1)[:, np.newaxis]
      cm_val   = cm_val.astype('float')   / cm_val.sum(axis=1)  [:, np.newaxis]
      cm_test  = cm_test.astype('float')  / cm_test.sum(axis=1) [:, np.newaxis]
        # Lists
    cms = [cm_train, cm_val, cm_test]
    titles = ['Train', 'Validation', 'Test']
        # Plot
    fig, axes = plt.subplots(nrows=1, ncols=3, dpi=dpi, figsize=(15, 8))
    #fig.suptitle(title)
    for i, ax in enumerate(axes):
      im = ax.imshow(cms[i], interpolation='nearest', cmap=cmap, vmin=0, vmax=1)
      # Label
      ax.set(xticks=np.arange(cms[i].shape[1]),
             yticks=np.arange(cms[i].shape[0]),
             xticklabels=classes,
             yticklabels=classes,
             title=titles[i])
      # Rotate the tick labels and set their alignment.
      plt.setp(ax.get_yticklabels(), rotation=90, ha="right", rotation_mode="anchor")
    # Loop
    for c, cm in enumerate(cms):
        # Loop over data dimensions and create text annotations.
        fmt = '.2f' if normalize else 'd'
        thresh = cm.max() / 2.
        for i in range(cm.shape[0]):
            for j in range(cm.shape[1]):
                axes[c].text(j, i, format(cm[i, j], fmt),
                             ha="center", va="center",
                             color="white" if cm[i, j] > thresh else "black")
    # For only one ax
    axes[0].set(ylabel='True label',)
    axes[1].set(xlabel='Predicted label')
    # Adjust
    fig.subplots_adjust(right=0.8)
    cbar_ax = fig.add_axes([0.83, 0.315, 0.025, 0.375]) # [left, bottom, width, height]
    fig.colorbar(im, cax=cbar_ax)
    # Plot
    plt.show()


def weak_aug(self, p=0.5):
    '''Create a weakly augmented image framework'''
    return A.Compose([
            A.HorizontalFlip(),
            A.OneOf([A.IAAAdditiveGaussianNoise(), A.GaussNoise(),], p=0.2),
            A.OneOf([A.MotionBlur(p=0.2), A.MedianBlur(blur_limit=3, p=0.1),], p=0.2),
            A.ShiftScaleRotate(shift_limit=0.0625, scale_limit=0.2, rotate_limit=10, p=0.2),
            A.OpticalDistortion(p=0.2),
            A.OneOf([A.CLAHE(clip_limit=2), A.IAASharpen(), A.IAAEmboss(),], p=0.3),
        ], p=p)

def strong_aug(self, p=0.5):
    '''Create a strongly augmented image framework'''
    return A.Compose([
            #RandomRotate90(), #Flip(), #Transpose(),
            A.HorizontalFlip(),
            A.OneOf([A.IAAAdditiveGaussianNoise(), A.GaussNoise(),], p=0.2),
            A.OneOf([A.MotionBlur(p=0.2), A.MedianBlur(blur_limit=3, p=0.1),], p=0.2),
            A.ShiftScaleRotate(shift_limit=0.0625, scale_limit=0.2, rotate_limit=45, p=0.2),
            A.OneOf([A.OpticalDistortion(p=0.2), A.GridDistortion(p=0.1), A.IAAPiecewiseAffine(p=0.2),], p=0.2),
            A.OneOf([A.CLAHE(clip_limit=2), A.IAASharpen(), A.IAAEmboss(), A.RandomBrightnessContrast(),], p=0.3),
            A.HueSaturationValue(p=0.3),
        ], p=p)


def augment_images(self, img, aug_factor):
    '''Augment and return images'''
    augmented_images = []
    my_weak_aug = self.weak_aug(p=1)
    my_strong_aug = self.strong_aug(p=1)

    # Augment and standardise
    if aug_factor > 1:
      for i in range(aug_factor):
        if i < int(aug_factor*0.7):
          augmented_images.append(my_weak_aug(image=img)['image'])
        else:
          augmented_images.append(my_strong_aug(image=img)['image'])

      augmented_images = np.array(augmented_images)
      return np.concatenate((img[None,:,:,:], augmented_images))
    else:
      return img

def process_data(self, raw_data_list, augs):
    '''Convert images to RGB, equalise, augment and standardise.'''
    processed_data = []
    batch = []
    for i, raw_data in enumerate(raw_data_list):
      for img in raw_data:
        img = self.convertToRGB(img.astype('uint8'))
        img = self.equalise_image(img, eq_type='HSV')
        imgs = self.augment_images(img, augs[i])
        imgs = self.standardise_images(imgs)
        batch.append(imgs)
        processed_data.append(np.concatenate(batch))
        batch = []
    return processed_data

def make_fpr_tpr_auc_dicts(self, y, probs_list):
    '''Compute and return the ROC curve and ROC area for each class in dictionaries'''
    # Dicts
    fpr = dict()
    tpr = dict()
    thresholds = dict()
    roc_auc = dict()
    # For test
    for i in range(self.NUM_CLASSES):
        fpr[i], tpr[i], thresholds[i] = metrics.roc_curve(y[:, i], probs_list[:, i])
        roc_auc[i] = metrics.auc(fpr[i], tpr[i])
    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = metrics.roc_curve(y.ravel(), probs_list.ravel())
    roc_auc["micro"] = metrics.auc(fpr["micro"], tpr["micro"])
    # Compute macro-average ROC curve and ROC area
    # First aggregate all false positive rates
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(self.NUM_CLASSES)]))
    # Then interpolate all ROC curves at this points
    mean_tpr = np.zeros_like(all_fpr)
    for i in range(self.NUM_CLASSES):
        mean_tpr += interp(all_fpr, fpr[i], tpr[i])
    # Finally average it and compute AUC
    mean_tpr /= self.NUM_CLASSES
    fpr["macro"] = all_fpr
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = metrics.auc(fpr["macro"], tpr["macro"])
    return fpr, tpr, thresholds, roc_auc

def randomColorGenerator(self, number_of_colors=1):
    return ['#'+''.join([np.random.choice(list('0123456789ABCDEF')) for j in range(6)]) for i in range(number_of_colors)]

def plot_roc_auc_curves(self, fpr, tpr, roc_auc, xlim=(-0.0025, 0.03), ylim=(0.99, 1.001)):
    '''Plot ROC AUC Curves'''
    fig,axes = plt.subplots(nrows=1, ncols=2, dpi=150, figsize=(10,5))
    # Define
    lw = 2.5
    axes[0].set_xlabel('False Positive Rate')
    axes[1].set_xlabel('False Positive Rate')
    axes[0].set_ylabel('True Positive Rate')
    # Loop
    for i in range(self.NUM_CLASSES):
      axes[0].plot(fpr[i], tpr[i], color=self.class_colors[i], label='{0} ({1:0.2f})' ''.format(self.num_to_class[i], roc_auc[i]*100))
      axes[1].plot(fpr[i], tpr[i], color=self.class_colors[i], lw=lw, label='{0} ({1:0.2f})' ''.format(self.num_to_class[i], roc_auc[i]*100))
    # Left plot
    axes[0].plot(fpr['micro'], tpr['micro'], label='Micro avg ({:0.2f}%)' ''.format(roc_auc['micro']*100), linestyle=':', color='deeppink')
    axes[0].plot(fpr['macro'], tpr['macro'], label='Macro avg ({:0.2f}%)' ''.format(roc_auc['macro']*100), linestyle=':', color='navy')
    axes[0].plot([0, 1], [0, 1], color='k', linestyle='--')
    axes[0].scatter(0,1, label='Ideal', s=2)
    # Right plot
    axes[1].plot(fpr['micro'], tpr['micro'], lw=lw, label='Micro avg ({:0.2f}%)'.format(roc_auc['micro']*100), linestyle=':', color='deeppink')
    axes[1].plot(fpr['macro'], tpr['macro'], lw=lw, label='Macro avg ({:0.2f}%)'.format(roc_auc['macro']*100), linestyle=':', color='navy')
```

D.3 My Utility Methods

```
    axes[1].plot([0, 1], [0, 1], color='k', linestyle='--')
    axes[1].scatter(0,1, label='Ideal', s=50)
    # Limits
    axes[1].set_xlim(xlim)
    axes[1].set_ylim(ylim)
    # Grids
    axes[0].grid(True, linestyle='dotted', alpha=1)
    axes[1].grid(True, linestyle='dotted', alpha=1)
    # Legends
    axes[0].legend(loc=4)
    axes[1].legend(loc=4)
    # Plot
    plt.legend(loc="lower right")
    plt.tight_layout()
    plt.show()

  def process_data(self, folder, y_class, return_compressed=True):
    '''Get image data from folder'''
    imgs = []
    for i, filename in enumerate(sorted(os.listdir(folder))):
      # We don't want .DS_Store or any other data files
      if filename.split(".")[-1].lower() in {"jpeg", "jpg", "png"}:
        # Load, Convert, Equalise, Standardise, Rescale
        img = cv2.imread(os.path.join(folder,filename))
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = self.equalise_image(img, eq_type='HSV')
        if return_compressed:
          img = self.standardise_image(img)
          img = self.resize_image(img)
        # Append
        if img is not None:
          imgs.append(img)
    # Concatenate
    if return_compressed:
      x_test = np.concatenate(imgs, axis=0)
      x_test = x_test.reshape(x_test.shape[0]//28, 28, 28, 3)
      y_test = np.array([y_class]*len(x_test))
    # Remove empty
    if return_compressed:
      indices_to_remove = []
      for i, x_i in enumerate(x_test):
        if x_i.std() == 0:
          indices_to_remove.append(i)
      x_test = np.delete(x_test, indices_to_remove, axis=0)
      y_test = np.delete(y_test, indices_to_remove, axis=0)
      # One hot encode if necessary
      if y_test.ndim == 1:
        y_test = self.one_hot_encode(y_test)
      return x_test, y_test
    return np.array(imgs)

  def get_detected_faces_cv(self, image, image_copy, scaleFactor = 1.1, cascade_path='data/haarcascades/haarcascade_frontalface_alt2.xml'):
    '''Returns detected faces's region of interests and coordinates using OpenCV'''
    rois = []
    coordinates_list = []
    cascade = cv2.CascadeClassifier(cascade_path)
    faces_rect = cascade.detectMultiScale(image_copy, scaleFactor=scaleFactor, minNeighbors=5)
    for i,(x, y, w, h) in enumerate(faces_rect):
      coordinates_list.append((x, y, w, h))
      rois.append(image_copy[y:y+h, x:x+w])
    return rois, coordinates_list

  def get_detected_faces_mtcnn(self, image, image_copy, detector):
    '''Returns detected faces's region of interests and coordinates using MTCNN'''
    rois = []
    coordinates_list = []
    faces_rect = detector.detect_faces(image_copy)
    for face in faces_rect:
      x, y, w, h = face['box']
      coordinates_list.append((x, y, w, h))
      rois.append(image_copy[y:y+h, x:x+w])
    return rois, coordinates_list

  def tag_images(self, images, all_faces, preds_reco, preds_fer, preds_gender, y_hat_prob_fer,
                 coordinates_list, fontFace=cv2.FONT_HERSHEY_SIMPLEX, thickness=5):
    '''Tags faces according to predicted class on the given image'''
    c = 0                                 # Counter
    for i, img in enumerate(images):      # For each image
      for j in range(len(all_faces[i])):  # For each face in each image
        # Calculate variables
        imageWidth, imageHeight, _ = img.shape
        x, y, w, h = coordinates_list[i][j]
        myColor = tuple([x_in*255 for x_in in mc.to_rgb(class_colors[preds_reco[c]])])
        myCoordinates = coordinates_list[i][j][0], coordinates_list[i][j][1]
        myLabelText = num_to_class_fer[preds_fer[c]]+' '+str(y_hat_prob_fer[c])
        myFontScale = (imageWidth * imageHeight) / (1000 * 1000)
        myFaceCenter = (int(x+w/2), int(y+h/2))
        myRadius = int(h/2)
        # Draw on image - rectangle for M, circle for F
        if preds_gender[c] == 0:
          rect = cv2.rectangle(img, (x, y), (x+w, y+h), myColor, thickness)
        else:
          circle = cv2.circle(img, myFaceCenter, myRadius, myColor, thickness)
        cv2.putText(img, org=(myCoordinates), text=myLabelText, fontFace=fontFace,
                    fontScale=myFontScale,color=(0,255,0), thickness=thickness)
        # Increment counter
        c += 1

  def load_final_test_images(self, folder, detection_method='cv'):
    '''Load the additional test images.'''
    # Lists
    pics = []
    copies = []
    all_faces = []
    coordinates_list = []
    normalised_faces = []
    unnormalised_faces = []
    # Method type
    #if detection_method == 'mtcnn':
    #  detector = MTCNN(min_face_size=50)
    # Get images
    print('Getting images...')
    for filename in sorted(os.listdir(folder)):
      pic_of_interest = cv2.imread(os.path.join(folder, filename))
      pic_of_interest = self.convertToRGB(pic_of_interest)
      pics.append(pic_of_interest)
    # Get face coordinates
    print('Getting face coordinates...')
    for pic in pics:
      image_copy = pic.copy()
      copies.append(image_copy)
      # Type
      #if detection_method == 'cv':
      faces_batch_temp, coordinates_temp = self.get_detected_faces_cv(pic, image_copy)
      #else:
      #  faces_batch_temp, coordinates_temp = self.get_detected_faces_mtcnn(pic, image_copy, detector)
      # Find fake-faces
      indices_to_remove = []
      for i, face in enumerate(faces_batch_temp):
        if face.size == 0:
          indices_to_remove.append(i)
      # Remove
      faces_batch = np.delete(faces_batch_temp, indices_to_remove, axis=0)
      coordinates = np.delete(coordinates_temp, indices_to_remove, axis=0)
      # Append
      all_faces.append(faces_batch)
      coordinates_list.append(coordinates)
    # Convert faces
    print('Processing faces...')
    for face_batch in all_faces:
      for i, face in enumerate(face_batch):
        if face.size != 0:
          unnormalised_faces.append(face)
          face = self.equalise_image(face, eq_type='HSV')
          face = self.standardise_image(face)
          normalised_faces.append(self.resize_image(face))
    print('Done!')
    return pics, copies, all_faces, coordinates_list, normalised_faces, unnormalised_faces

  def plot_before_after_tag(self, images, copies):
    '''Plot the before and after tag images side by side'''
    for i, img in enumerate(images):
      plt.figure(dpi=150)
      plt.imshow(np.hstack((copies[i], img)))
      plt.axis('off')
      plt.show()

  def restore_model(self, graph, graph_dir, checkpoint_dir):
    '''Import graph and restore model variables'''
    with graph.as_default():
      # Load graph and restore tf variables
      saver = tf.train.import_meta_graph(graph_dir)
      latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
      sess = tf.Session(graph=graph)
      saver.restore(sess, latest_checkpoint)
      # Get relevant tensors
      tf_cnn_softmax = graph.get_tensor_by_name('CNN/Softmax:0')
      tf_placeholder = graph.get_tensor_by_name('Placeholder:0')
    return sess, tf_cnn_softmax, tf_placeholder

  def run_model(self, sess, tf_placeholder, tf_cnn_softmax, x_test):
    '''Run model'''
    probs = sess.run(tf_cnn_softmax, feed_dict={tf_placeholder: x_test})
    y_hat = np.argmax(probs, axis=1)
    return probs, y_hat

  def run_keras_model(self, model, x_test):
    '''Run Keras Model'''
    probs = model.predict(np.array(x_test))
    y_hat = np.argmax(probs, axis=1)
    return probs, y_hat

  def restore_keras_graph(self, graph_dir_keras_fer, checkpoint_dir_keras_fer):
    '''Check if Keras model's graph is stored'''
    # For the sake of consistency, we also store Keras model's as checkpoint so we can use it as TF
    if not os.path.isfile('Keras_logdir/model.ckpt.meta'):
      model_fer = keras.models.load_model(graph_dir_keras_fer)
      saver_fer = tf.train.Saver()
      sess_fer = keras.backend.get_session()
      save_path = saver_fer.save(sess_fer, checkpoint_dir_keras_fer)
```

D.4 My CNN

```python
import numpy as np
import tensorflow as tf
from sklearn.utils import shuffle
from myutilitymethods import MyMethods

class MyCNN:

  """
  Convolutional neural network class which trains on data for facial tasks,
  such as face recognition, or gender classification.

  Parameters
  ----------
  x_train          : Train images data
  y_train          : Train images labels
  x_test           : Test images data
  y_test           : Test images labels
  output_dir       : Directory for saving checkpoints
  num_to_class     : List for converting a number to a class value string
  class_to_num     : Dictionary for converting a class string to a number
  lr               : Learning rate                        (default = 1e-3)
  nb_epochs        : Number of epochs                     (default = 10)
  batch_size_train : Train batch size                     (default = 30)
  seed             : Seed for random functions            (default = 0)
  type             : Type of model                        (default = softmax)
  """

  def __init__(self,
         x_train,
         y_train,
         x_test,
         y_test,
         output_dir,
         num_to_class,
         class_to_num,
         lr=1e-3,
         nb_epochs=10,
         batch_size_train=30,
         seed=0,
         final_activation='softmax'):

        # Variables
    self.x_train = x_train
    self.y_train = y_train
    self.x_test = x_test
    self.y_test = y_test
    self.output_dir = output_dir
    self.lr = lr
    self.nb_epochs = nb_epochs
    self.batch_size_train = batch_size_train
    self.seed = seed
    self.final_activation = final_activation.lower()

    self.nb_classes = y_train.shape[1] # This assumes one-hot encode
    self.nb_images, self.edge, _, self.channels = x_train.shape
    self.nb_train_iterations = self.nb_images // batch_size_train
    self.im = tf.placeholder(tf.float32, [None, 28, 28, self.channels])
    self.labels = tf.placeholder(tf.float32, [None, self.nb_classes])

    self.accuracy_list = []
    self.losses_list = []
    self.test_losses_list = []
    self.test_accuracy_list = []
    self.val_accuracy_list = []
    self.val_losses_list = []
    self.preds_list = []
    self.probs_list = []
    self.preds_list_test = []
    self.probs_list_test = []

    self.mm = MyMethods(self.nb_classes, num_to_class, class_to_num)
    self.sess = tf.InteractiveSession()
    self.summary = tf.Summary()
    self.writer = tf.summary.FileWriter(self.output_dir)
    self.writer.add_graph(self.sess.graph)

  def model_variables(self):
    '''Print out all class variables'''
    print('x_train             :', self.x_train.shape)
    print('y_train             :', self.y_train.shape)
    print('x_test              :', self.x_test.shape)
    print('y_test              :', self.y_test.shape)
    print('output_dir          :', self.output_dir)
    print('lr                  :', self.lr)
    print('nb_epochs           :', self.nb_epochs)
    print('batch_size_train    :', self.batch_size_train)
    print('seed                :', self.seed)
    print('nb_classes          :', self.nb_classes)
    print('nb_images           :', self.nb_images)
    print('nb_train_iterations :', self.nb_train_iterations)
    print('im                  :', self.im)
    print('labels              :', self.labels)


  def set_up_saver(self):
    self.saver = tf.train.Saver()

  def create_model(self):
    '''Creates defined TensorFlow model'''
    with tf.variable_scope('CNN', reuse=tf.AUTO_REUSE):
      self.first_layer = tf.layers.conv2d(self.im, 32, (4,4), strides=1, activation=tf.nn.relu)
      self.second_layer = tf.layers.conv2d(self.first_layer, 64, (4,4), strides=2, activation=tf.nn.relu)
      self.third_layer = tf.layers.conv2d(self.second_layer, 128, (4,4), strides=2, activation=tf.nn.relu)
      self.fourth_layer = tf.layers.conv2d(self.third_layer, 256, (4,4), strides=2, activation=None)
      self.flattened = tf.layers.flatten(self.fourth_layer)
      self.logits = tf.layers.dense(self.flattened, self.nb_classes, activation=None)
      if self.final_activation == 'softmax':
        self.preds = tf.nn.softmax(self.logits)
      elif self.final_activation == 'sigmoid':
        self.preds = tf.nn.sigmoid(self.logits)
      else:
        print("Error: the model.final_activation parameter can only either be 'softmax' or 'sigmoid'. ")

  def model_summary(self):
    '''Prints out the model's architecture shapes'''
    print('first_layer  : conv2D  ->', self.first_layer.shape)
    print('second_layer : conv2D  ->', self.second_layer.shape)
    print('third_layer  : conv2D  ->', self.third_layer.shape)
    print('fourth_layer : conv2D  ->', self.fourth_layer.shape)
    print('flattened.   : Flatten ->', self.flattened.shape)
    print('logits       : Dense   ->', self.logits.shape)
    print('preds        : Softmax ->', self.preds.shape)

  def compute_loss(self):
    '''Computes loss for the model'''
    with tf.variable_scope('loss'):
      if self.final_activation == 'softmax':
        self.loss = tf.losses.softmax_cross_entropy(self.labels, logits=self.logits)
        self.loss_summ = tf.summary.scalar("softmax_loss", self.loss)
      elif self.final_activation == 'sigmoid':
        self.loss = tf.losses.sigmoid_cross_entropy(self.labels, logits=self.logits)
        self.loss_summ = tf.summary.scalar("sigmoid_loss", self.loss)
      else:
        print("Error: the model.final_activation parameter can only either be 'softmax' or 'sigmoid'. ")

  def optimizer(self):
    '''Defines optimizer used for the model'''
    with tf.variable_scope('optimizer'):
      optimizer = tf.train.AdamOptimizer(learning_rate=self.lr, beta1=0.5)
      self.model_vars = tf.trainable_variables()
      self.trainer = optimizer.minimize(self.loss, var_list=self.model_vars)

  def train(self, verbose=False, cross_k_fold_validation=True):
    '''
    Trains the model on the parsed x_train and y_train values for the given number of epochs.
    Saves checkpoints in output directory.
    '''

    if cross_k_fold_validation:
      # We split the train data into k equal subsets for k-fold cross validation
      # Note: these subsets are different from iteration batches used for optimisated gradient descent
      cut_x_train = self.x_train.reshape(self.x_train.shape[0], np.prod(self.x_train.shape[1:])) # (20000, 2352)
      subset_size = cut_x_train.shape[0]//self.nb_epochs
      cut_x_train = cut_x_train.reshape(self.nb_epochs, cut_x_train.shape[1]*subset_size) # (200, 235200)

      cut_y_train = np.argmax(self.y_train, axis=1) # (20000,)
      cut_y_train = cut_y_train.reshape(self.nb_epochs, subset_size) # (200, 100)

    else:
      self.x_train, self.y_train, self.x_val, self.y_val = self.mm.split_train_test(self.x_train, self.y_train)

      x_in = self.x_train
      y_in = self.y_train

      if self.y_val.ndim == 1:
        self.y_val = self.mm.one_hot_encode(self.y_val)

    for epoch in range(self.nb_epochs):

      if cross_k_fold_validation:
        # Get validation data from the training set, both which change at each epoch
        x_in = np.delete(cut_x_train, epoch, axis=0) # (199, 235200)
        y_in = np.delete(cut_y_train, epoch, axis=0) # (199, 100)
        self.x_val = cut_x_train[epoch] # (1, 235200) = (235200,)
        self.y_val = cut_y_train[epoch] # (1, 100) = (100,)

        # Reshape back to standard
        x_in = x_in.reshape(x_in.shape[0]*subset_size, x_in.shape[1]//subset_size) # (19900, 2352)
        x_in = x_in.reshape(x_in.shape[0], 28, 28, self.channels) # (19900, 28, 28, 3)

        y_in = y_in.reshape(y_in.shape[0]*subset_size) # (19900,)
        if y_in.ndim == 1:
          y_in = self.mm.one_hot_encode(y_in) # (19900, 4)

        self.x_val = self.x_val.reshape(subset_size, 28, 28, self.channels) # (100, 28, 28, 3)
        if self.y_val.ndim == 1:
```

D.5 My CNN

```
        self.y_val = self.mm.one_hot_encode(self.y_val) # (100, 4)

        # Reset
        self.preds_list = []
        self.probs_list = []
        self.probs_list_val = []
        self.preds_list_val = []

        # Shuffle order for both subsets
        x_in, y_in = shuffle(x_in, y_in, random_state=self.seed)
        self.x_val, self.y_val = shuffle(self.x_val, self.y_val, random_state=self.seed)

        # Loop through iterations
        for i in range(self.nb_train_iterations):

            # Take batch
            input_x_train = x_in[i*self.batch_size_train: (i+1)*self.batch_size_train]
            input_y_train = y_in[i*self.batch_size_train: (i+1)*self.batch_size_train]

            # Train model
            _, probs, loss, loss_summ = self.sess.run([self.trainer, self.preds, self.loss, self.loss_summ],
                                                      feed_dict={self.im: input_x_train, self.labels: input_y_train})

            self.writer.add_summary(loss_summ, epoch * self.nb_train_iterations + i)

        # We want the entire train set score after each epoch for those epoch values
        probs, loss = self.sess.run([self.preds, self.loss], feed_dict={self.im: x_in, self.labels: y_in})
        y_hat = np.argmax(probs, axis=1)

        self.probs_list.append(probs)
        self.preds_list.append(y_hat)

        y_real = np.argmax(y_in, axis=1)
        acc_train = np.mean(self.preds_list == y_real)
        self.accuracy_list.append(acc_train)
        self.losses_list.append(loss)

        # We want the entire validation set score after each epoch
        probs_val, loss_val = self.sess.run([self.preds, self.loss], feed_dict={self.im: self.x_val, self.labels: self.y_val})
        y_hat_val = np.argmax(probs_val, axis=1)
        y_real_val = np.argmax(self.y_val, axis=1)

        self.probs_list_val.append(probs_val)
        self.preds_list_val.append(y_hat_val)

        acc_val = np.mean(y_hat_val == y_real_val)
        self.val_accuracy_list.append(acc_val)
        self.val_losses_list.append(loss_val)

        # Print
        if verbose:
            print(f'Epoch {epoch}, Train acc {np.round(acc_train, 2)}, Validation acc {np.round(acc_val, 2)}')

        # Save
        self.saver.save(self.sess, self.output_dir, global_step=epoch)

    # For the final epoch we want the predictions for the non-shuffled x_train and y_train (not x_in and y_in)
    probs = self.sess.run(self.preds, feed_dict={self.im: self.x_train, self.labels: self.y_train})
    self.probs_list = probs
    self.preds_list = np.argmax(probs, axis=1)

    # Flatten
    self.probs_list_val = np.concatenate(self.probs_list_val, axis=0)
    self.preds_list_val = np.concatenate(self.preds_list_val, axis=0)

def test(self):
    '''Tests and prints the model's accuracy on the test data set or a custom input.'''

    # Reset
    self.preds_list_test = []
    self.probs_list_test = []

    # Run
    probs_test = self.sess.run(self.preds, feed_dict={self.im: self.x_test, self.labels: self.y_test})

    # Append results
    self.probs_list_test.append(probs_test)
    self.preds_list_test.append(np.argmax(probs_test, axis=1))

    # Flatten
    self.probs_list_test = np.concatenate(self.probs_list_test, axis=0)
    self.preds_list_test = np.concatenate(self.preds_list_test, axis=0)

    # Print final accuracy
    y_real_test = np.argmax(self.y_test, axis=1)
    full_acc_test = np.mean(self.preds_list_test==y_real_test)
    print('Test accuracy achieved: %.3f' %full_acc_test)
```

D.6 My Deep CNN

```python
import keras
import numpy as np
import tensorflow as tf
from sklearn.utils import shuffle
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.losses import categorical_crossentropy
from keras.optimizers import Adam
from keras.regularizers import l2
from keras.callbacks import ReduceLROnPlateau, TensorBoard, EarlyStopping, ModelCheckpoint

class MyDeepCNN:

  """
  Convolutional neural network class which trains on data for facial tasks,
  such as face recognition, or gender classification.

  Parameters
  ----------
  x_train          : Train images data
  y_train          : Train images labels
  x_test           : Validation images data
  y_test           : Validation images labels
  x_test           : Test images data
  y_test           : Test images labels
  output_dir       : Directory for saving model
  lr               : Learning rate                    (default = 1e-3)
  beta_1           : Beta 1                           (default = 0.9)
  beta_2           : Beta 2                           (default = 0.999)
  nb_epochs        : Number of epochs                 (default = 50)
  epsilon          : Epsilon                          (default = 1e-7)
  batch_size       : Train batch size                 (default = 64)
  seed             : Seed for random functions        (default = 0)
  num_features     : Number of features               (default = 64)
  """

  def __init__(self,
        x_train,
        y_train,
        x_val,
        y_val,
        x_test,
        y_test,
        output_dir,
        lr=1e-3,
        beta_1=0.9,
        beta_2=0.999,
        nb_epochs=50,
        epsilon=1e-7,
        batch_size=64,
        seed=0,
        num_features=64):

    # Variables
    self.x_train = x_train
    self.y_train = y_train
    self.x_val = x_val
    self.y_val = y_val
    self.x_test = x_test
    self.y_test = y_test

    self.output_dir = output_dir
    self.lr = lr
    self.beta_1 = 0.9
    self.beta_2 = 0.999
    self.nb_epochs = nb_epochs
    self.epsilon = epsilon
    self.batch_size = batch_size
    self.seed = seed
    self.num_features = num_features

    self.model = Sequential()
    self.nb_classes = y_train.shape[1] # This assumes one-hot encode
    self.nb_images, self.edge, _, self.channels = self.x_train.shape
    self.preds_list_test = []
    self.probs_list_test = []

    self.history = None
    self.sess = tf.InteractiveSession()
    keras.backend.set_session(self.sess)

  def model_variables(self):
    '''Print out all class variables'''
    print('x_train      :', self.x_train.shape)
    print('y_train      :', self.y_train.shape)
    print('x_val        :', self.x_val.shape)
    print('y_val.       :', self.y_val.shape)
    print('x_test.      :', self.x_test.shape)
    print('y_test.      :', self.y_test.shape)
    print('output_dir   :', self.output_dir)
    print('lr           :', self.lr)
    print('beta_1       :', self.beta_1)
    print('beta_2       :', self.beta_2)
    print('nb_epochs    :', self.nb_epochs)
    print('epsilon      :', self.epsilon)
    print('batch_size   :', self.batch_size)
    print('seed         :', self.seed)
    print('num_features :', self.num_features)
    print('nb_classes   :', self.nb_classes)
    print('nb_images    :', self.nb_images)

  def create_model(self):
    '''Creates defined TensorFlow model'''
    self.model.add(Conv2D(self.num_features, kernel_size=(3, 3), activation='relu', input_shape=(self.edge, self.edge, self.channels), kernel_regulari
    self.model.add(Conv2D(self.num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    self.model.add(BatchNormalization())
    self.model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    self.model.add(Dropout(0.5))

    self.model.add(Conv2D(2*self.num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    self.model.add(BatchNormalization())
    self.model.add(Conv2D(2*self.num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    self.model.add(BatchNormalization())
    self.model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    self.model.add(Dropout(0.5))

    self.model.add(Conv2D(2*2*2*self.num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    self.model.add(BatchNormalization())
    self.model.add(Conv2D(2*2*2*self.num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    self.model.add(BatchNormalization())
    self.model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    self.model.add(Dropout(0.5))

    self.model.add(Flatten())

    self.model.add(Dense(2*2*2*self.num_features, activation='relu'))
    self.model.add(Dropout(0.4))
    self.model.add(Dense(2*2*self.num_features, activation='relu'))
    self.model.add(Dropout(0.4))
    self.model.add(Dense(2*self.num_features, activation='relu'))
    self.model.add(Dropout(0.5))

    self.model.add(Dense(self.nb_classes, activation='softmax'))

  def model_summary(self):
    '''Print model summary'''
    print(self.model.summary())

  def train(self, verbose=False):
    '''Trains the model on the parsed x_train and y_train values for the given number of epochs. Saves final .h5 model in output directory.'''
    # Convert True/False to 1/0 integer
    verbose = int(verbose)

    # Compile
    self.model.compile(loss=categorical_crossentropy,
                optimizer=Adam(lr=self.lr,
                        beta_1=self.beta_1,
                        beta_2=self.beta_2,
                        epsilon=self.epsilon),
                metrics=['accuracy'])

    # Train
    self.history = self.model.fit(np.array(self.x_train), np.array(self.y_train),
                    batch_size=self.batch_size,
                    epochs=self.nb_epochs,
                    verbose=verbose,
                    validation_data=(np.array(self.x_val), np.array(self.y_val)),
                    shuffle=True)

    # Save
    self.model.save(self.output_dir+'/my_keras_model.h5')
    self.model.save_weights(self.output_dir+'/my_keras_model_weights.h5')

  def test(self, verbose=False):
    '''Tests and prints the model's accuracy on the test data set.'''
    # Convert True/False to 1/0 integer
    verbose = int(verbose)

    # Reset
    self.preds_list_test = []
    self.probs_list_test = []

    # Run
    probs_test = self.model.predict(self.x_test, batch_size=self.batch_size, verbose=verbose)

    # Results
    self.probs_list_test.append(probs_test)
    self.preds_list_test = np.argmax(probs_test, axis=1)
    y_real_test = np.argmax(self.y_test, axis=1)

    # Print
    full_acc_test = np.mean(self.preds_list_test==y_real_test)
    print('Test accuracy achieved: %.3f' %full_acc_test)
```

D.7 End-to-End Model

```
import os
import sys
import cv2
import time
import keras
import random
import logging
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
import albumentations as A
from IPython import display
from datetime import datetime
from mtcnn.mtcnn import MTCNN
import matplotlib.colors as mc
from myutilitymethods import MyMethods
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.decomposition import PCA
from matplotlib.patches import Rectangle
from matplotlib.lines import Line2D
from sklearn.metrics import classification_report
from sklearn.utils.multiclass import unique_labels
from sklearn.metrics import confusion_matrix as sk_cm
from keras.regularizers import l2
from keras.optimizers import Adam
from keras.models import Sequential
from keras.losses import categorical_crossentropy
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.callbacks import ReduceLROnPlateau, TensorBoard, EarlyStopping, ModelCheckpoint


# ========== METHODS ==========
def process_data(folder, y_class, return_compressed=True):
    '''Get image data from folder'''
    imgs = []

    for i, filename in enumerate(sorted(os.listdir(folder))):

        # We don't want .DS_Store or any other data files
        if filename.split(".")[-1].lower() in {"jpeg", "jpg", "png"}:

            # Load, Convert, Equalise, Standardise, Rescale
            img = cv2.imread(os.path.join(folder,filename))
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img = mm.equalise_image(img, eq_type='HSV')

            if return_compressed:
                img = mm.standardise_image(img)
                img = mm.resize_image(img)

            # Append
            if img is not None:
                imgs.append(img)

    # Concatenate
    if return_compressed:
        x_test = np.concatenate(imgs, axis=0)
        x_test = x_test.reshape(x_test.shape[0]//28, 28, 28, 3)
        y_test = np.array([y_class]*len(x_test))

    # Remove empty
    if return_compressed:
        indices_to_remove = []
        for i, x_i in enumerate(x_test):
            if x_i.std() == 0:
                indices_to_remove.append(i)
        x_test = np.delete(x_test, indices_to_remove, axis=0)
        y_test = np.delete(y_test, indices_to_remove, axis=0)

        if y_test.ndim == 1:
            y_test = mm.one_hot_encode(y_test)

        return x_test, y_test

    return np.array(imgs)

def get_detected_faces_cv(image, image_copy, scaleFactor = 1.1, cascade_path='data/haarcascades/haarcascade_frontalface_alt2.xml'):
    '''Returns detected faces's region of interests and coordinates using OpenCV'''
    roia = []
    coordinates_list = []
    cascade = cv2.CascadeClassifier(cascade_path)
    faces_rect = cascade.detectMultiScale(image_copy, scaleFactor=scaleFactor, minNeighbors=5)
    for i,(x, y, w, h) in enumerate(faces_rect):
        coordinates_list.append((x, y, w, h))
        roia.append(image_copy[y:y+h, x:x+w])
    return roia, coordinates_list

def get_detected_faces_mtcnn(image, image_copy, detector):
    '''Returns detected faces's region of interests and coordinates using MTCNN'''
    rois = []
    coordinates_list = []
    faces_rect = detector.detect_faces(image_copy)
    for face in faces_rect:
        x, y, w, h = face['box']
        coordinates_list.append((x, y, w, h))
        rois.append(image_copy[y:y+h, x:x+w])
    return rois, coordinates_list

def tag_images(images, all_faces, preds_reco, preds_fer, preds_gender, y_hat_prob_fer, coordinates_list, fontFace=cv2.FONT_HERSHEY_SIMPLEX, thickness=
    '''Tags faces according to predicted class on the given image'''

    c = 0                               # Counter
    for i, img in enumerate(images):    # For each image
        for j in range(len(all_faces[i])):  # For each face in each image

            # Calculate variables
            imageWidth, imageHeight, _ = img.shape
            x, y, w, h = coordinates_list[i][j]
            myColor = tuple([x_in*255 for x_in in mc.to_rgb(class_colors[preds_reco[c]])])
            myCoordinates = coordinates_list[i][j][0], coordinates_list[i][j][1]
            myLabelText = num_to_class_fer[preds_fer[c]]+' '+str(y_hat_prob_fer[c])
            myFontScale = (imageWidth * imageHeight) / (1000 * 1000)
            myFaceCenter = (int(x+w/2), int(y+h/2))
            myRadius = int(h/2)

            # Draw on image - rectangle for M, circle for F
            if preds_gender[c] == 0:
                rect = cv2.rectangle(img, (x, y), (x+w, y+h), myColor, thickness)
            else:
                circle = cv2.circle(img, myFaceCenter, myRadius, myColor, thickness)

            cv2.putText(img, org=(myCoordinates), text=myLabelText, fontFace=fontFace,
                        fontScale=myFontScale,color=(0,255,0), thickness=thickness)

            # Increment counter
            c += 1

def recreate_keras_model(num_features, nb_classes, edge, channels):
    '''Re-creates Keras model'''
    model = keras.Sequential()

    model.add(Conv2D(num_features, kernel_size=(3, 3), activation='relu', input_shape=(edge, edge, channels), kernel_regularizer=l2(0.01)))
    model.add(Conv2D(num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(2*2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(2*2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Flatten())

    model.add(Dense(2*2*2*num_features, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(2*2*num_features, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(2*num_features, activation='relu'))
    model.add(Dropout(0.5))

    model.add(Dense(nb_classes, activation='softmax'))

    return model

def load_final_test_images(folder='TestImages/', detection_method='cv'):

    pics = []
    copies = []
    all_faces = []
    coordinates_list = []
    normalised_faces = []
    unnormalised_faces = []

    if detection_method == 'mtcnn':
        detector = MTCNN(min_face_size=50)

    # Get images
    print('Getting images...')
    for filename in sorted(os.listdir(folder)):
        pic_of_interest = cv2.imread(os.path.join(folder, filename))
        pic_of_interest = mm.convertToRGB(pic_of_interest)
        pics.append(pic_of_interest)

    # Get face coordinates
    print('Getting face coordinates...')
    for pic in pics:
        image_copy = pic.copy()
        copies.append(image_copy)

        if detection_method == 'cv':
            faces_batch_temp, coordinates_temp = get_detected_faces_cv(pic, image_copy)
        else:
            faces_batch_temp, coordinates_temp = get_detected_faces_mtcnn(pic, image_copy, detector)

        # Find fake-faces
        indices_to_remove = []

        for i, face in enumerate(faces_batch_temp):
            if face.size == 0:
                indices_to_remove.append(i)

        # Remove
        faces_batch = np.delete(faces_batch_temp, indices_to_remove, axis=0)
        coordinates = np.delete(coordinates_temp, indices_to_remove, axis=0)

        # Append
        all_faces.append(faces_batch)
        coordinates_list.append(coordinates)

    # Convert faces
    print('Processing faces...')
```

D.8 End-to-End Model

```
    for face_batch in all_faces:
        for i, face in enumerate(face_batch):
            if face.size != 0:
                unnormalised_faces.append(face)
                face = mm.equalise_image(face, eq_type='HSV')
                face = mm.standardise_image(face)
                normalised_faces.append(mm.resize_image(face))

    print('Done!')
    return pics, copies, all_faces, coordinates_list, normalised_faces, unnormalised_faces

def plot_before_after_tag(images, copies):

    for i, img in enumerate(images):
        plt.figure(dpi=150)
        plt.imshow(np.hstack((copies[i], img)))
        plt.axis('off')
        plt.tight_layout()
        plt.legend(lines, ['Myself', 'Sister', 'Mother', 'Father'],
                   bbox_to_anchor=(0.0, 1.15), loc="upper left", ncol=4)
        #rand_name = str(np.random.randint(1, 100000))
        #plt.savefig(f'{rand_name}.pdf', bbox_inches='tight', format='pdf', dpi=200)
        plt.show()

def restore_model(graph, graph_dir, checkpoint_dir):
    '''Import graph and restore model variables'''
    with graph.as_default():

        # Load graph and restore tf variables
        saver = tf.train.import_meta_graph(graph_dir)
        latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
        sess = tf.Session(graph=graph)
        saver.restore(sess, latest_checkpoint)

        # Get relevant tensors
        tf_cnn_softmax = graph.get_tensor_by_name('CNN/Softmax:0')
        tf_placeholder = graph.get_tensor_by_name('Placeholder:0')

    return sess, tf_cnn_softmax, tf_placeholder

def run_model(sess, tf_placeholder, tf_cnn_softmax, x_test):
    '''Run model'''
    probs = sess.run(tf_cnn_softmax, feed_dict={tf_placeholder: x_test})
    y_hat = np.argmax(probs, axis=1)
    return probs, y_hat

def run_keras_model(model, x_test):
    '''Run Keras Model'''
    probs = model.predict(np.array(x_test))
    y_hat = np.argmax(probs, axis=1)
    return probs, y_hat

def restore_keras_graph(graph_dir_keras_fer, checkpoint_dir_keras_fer):
    '''Check if Keras model's graph is stored'''
    # For the sake of consistency, we also store Keras model's
    # as checkpoint so we can use it as TF
    if not os.path.isfile('Keras_logdir/model.ckpt.meta'):
        model_fer = keras.models.load_model(graph_dir_keras_fer)
        saver_fer = tf.train.Saver()
        sess_fer = keras.backend.get_session()
        save_path = saver_fer.save(sess_fer, checkpoint_dir_keras_fer)

# ========== DEFINE VARIABLES ==========
NUM_CLASSES = 4
class_colors = ['C1', 'C2', 'C3', 'C0']
num_to_class = ['Myself', 'Sister', 'Mother', 'Father']
class_to_num = {'Myself': 0, 'Sister': 1, 'Mother': 2, 'Father': 3,}

NUM_CLASSES_FER = 7
num_to_class_fer = ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad', 'surprise']
class_to_num_fer = {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3, 'neutral':4, 'sad':5, 'surprise':6}

mm = MyMethods(NUM_CLASSES, num_to_class, class_to_num)
mm_fer = MyMethods(NUM_CLASSES_FER, num_to_class_fer, class_to_num_fer)

lines = [Line2D([0], [0], color=c) for c in class_colors]

# ========== MAIN ==========
def main():

    # Create new graphs
    graph_reco = tf.Graph()
    graph_fer = tf.Graph()
    graph_gender = tf.Graph()

    # Restore Reco model
    graph_dir_reco = 'Home_logdir/-149.meta'
    checkpoint_dir_reco = 'Home_logdir/'
    sess_reco, tf_cnn_softmax_reco, tf_placeholder_reco = restore_model(graph_reco, graph_dir_reco, checkpoint_dir_reco)

    # Restore Keras FER model
    weights_dir_keras_fer = 'Keras_logdir/my_keras_model_weights.h5'
    model_fer = recreate_keras_model(64, NUM_CLASSES_FER, 28, 3)
    model_fer.load_weights(weights_dir_keras_fer)

    # Restore Gender model
    graph_dir_gender = 'Gender_logdir/-74.meta'
    checkpoint_dir_gender = 'Gender_logdir/'
    sess_gender, tf_cnn_softmax_gender, tf_placeholder_gender = restore_model(graph_gender, graph_dir_gender, checkpoint_dir_gender)

    #Test
    face_values = load_final_test_images(detection_method='cv')

    pics = face_values[0]
    copies = face_values[1]
    all_faces = face_values[2]
    coordinates_list = face_values[3]
    normalised_faces = face_values[4]
    unnormalised_faces = face_values[5]

    # Run
    probs_reco, y_hat_reco = run_model(sess_reco, tf_placeholder_reco, tf_cnn_softmax_reco, normalised_faces)
    probs_fer, y_hat_fer = run_keras_model(model_fer, normalised_faces)
    probs_gender, y_hat_gender = run_model(sess_gender, tf_placeholder_gender, tf_cnn_softmax_gender, normalised_faces)
    y_hat_prob_fer = np.round(np.max(probs_fer, axis=1), 2)

    # Tag and plot
    tag_images(pics, all_faces, y_hat_reco, y_hat_fer, y_hat_gender, y_hat_prob_fer, coordinates_list)
    plot_before_after_tag(pics, copies)

if __name__ == "__main__":
    main()
```

D.9 Real-Time Model

```python
import cv2
import keras
import argparse
import numpy as np
import tensorflow as tf
import matplotlib.colors as mc
import matplotlib.pyplot as plt
from keras.regularizers import l2
from keras.optimizers import Adam
from keras.models import Sequential
from mytilitymethods import MyMethods
from keras.losses import categorical_crossentropy
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.callbacks import ReduceLROnPlateau, TensorBoard, EarlyStopping, ModelCheckpoint

# ========== DEFINE VARIABLES ==========
NUM_CLASSES = 4
NUM_CLASSES_FER = 7

class_colors = ['C1', 'C2', 'C3', 'C0']
bgr_colors = [(0,165,255), (0,255,0), (0,0,255), (255,0,0)]

num_to_class_gender = ['M', 'F']
num_to_class = ['Myself', 'Sister', 'Mother', 'Father']
num_to_class_fer = ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad', 'surprise']

class_to_num = {'Myself': 0, 'Sister': 1, 'Mother': 2, 'Father': 3,}
class_to_num_fer = {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3, 'neutral':4, 'sad':5, 'surprise':6}

path = 'data/haarcascades/haarcascade_frontalface_alt2.xml'
mm = MyMethods(NUM_CLASSES, num_to_class, class_to_num)
mm_fer = MyMethods(NUM_CLASSES_FER, num_to_class_fer, class_to_num_fer)
fontFace=cv2.FONT_HERSHEY_SIMPLEX
thickness = 5
green_color = (0,255,0)

# ========== METHODS ==========
def recreate_keras_model(num_features, nb_classes, edge, channels):
    '''Re-creates Keras model'''
    model = keras.Sequential()

    model.add(Conv2D(num_features, kernel_size=(3, 3), activation='relu', input_shape=(edge, edge, channels), kernel_regularizer=l2(0.01)))
    model.add(Conv2D(num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(2*2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(2*2*2*num_features, kernel_size=(3, 3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(Dropout(0.5))

    model.add(Flatten())

    model.add(Dense(2*2*2*num_features, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(2*2*num_features, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(2*num_features, activation='relu'))
    model.add(Dropout(0.5))

    model.add(Dense(nb_classes, activation='softmax'))

    return model

# ========== MAIN ==========
def main():

    # Define Argsparser
    parser = argparse.ArgumentParser(description='')
    parser.add_argument('-a', '--allModels', action='store_true', default=True, help='Run all three models: face recognition, gender classification, and
    parser.add_argument('-fr', '--faceReco', action='store_true', default=False, help='Run only face recognition model.')
    parser.add_argument('-gc', '--genderClass', action='store_true', default=False, help='Run only gender classification model.')
    parser.add_argument('-ed', '--emotionDetect', action='store_true', default=False, help='Run only emotion detection model.')
    args = parser.parse_args()

    # XOR only
    if args.faceReco or args.emotionDetect or args.genderClass:
        args.allModels = False

    # Open OpenCV Camera
    cap = cv2.VideoCapture(0)
    print(cap.isOpened())

    # Restore graphs
    graph_reco = tf.Graph()
    graph_fer = tf.Graph()
    graph_gender = tf.Graph()

    #detector = MTCNN(min_face_size=50)

    # Restore reco model
    if args.allModels or args.faceReco:
        graph_dir_reco = 'Home_logdir/-149.meta'
        checkpoint_dir_reco = 'Home_logdir/'
        reco_values = mm.restore_model(graph_reco, graph_dir_reco, checkpoint_dir_reco)
        sess_reco = reco_values[0]
        tf_cnn_softmax_reco = reco_values[1]
        tf_placeholder_reco = reco_values[2]

    # Restore Keras FER model
    if args.allModels or args.emotionDetect:
        weights_dir_keras_fer = 'Keras_logdir/my_keras_model_weights.h5'
        graph_dir_keras_fer = 'Keras_logdir/my_keras_model.h5'
        checkpoint_dir_keras_fer = 'Keras_logdir/model.ckpt'
        #model_fer = keras.models.load_model(graph_dir_keras_fer)
        model_fer = recreate_keras_model(64, NUM_CLASSES_FER, 28, 3)
        model_fer.load_weights(weights_dir_keras_fer)

    # Restore Gender model
    if args.allModels or args.genderClass:
        graph_dir_gender = 'Gender_logdir/-74.meta'
        checkpoint_dir_gender = 'Gender_logdir/'
        gender_values = mm.restore_model(graph_gender, graph_dir_gender, checkpoint_dir_gender)
        sess_gender = gender_values[0]
        tf_cnn_softmax_gender = gender_values[1]
        tf_placeholder_gender = gender_values[2]

    frames = []

    # Run camera
    while(True):
        # Capture frame-by-frame
        ret, frame = cap.read()

        # Our operations on the frame come here
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        frames.append(rgb_frame)

        # Copy
        image_copy = rgb_frame.copy()

        # Get heads
        cascade = cv2.CascadeClassifier(path)
        faces_rect = cascade.detectMultiScale(rgb_frame, scaleFactor=1.1, minNeighbors=5)
        #faces_rect = detector.detect_faces(rgb_frame)

        # Plot rectangle(s) for each face in each frame
        for face in faces_rect:
            x, y, w, h = face
            #x, y, w, h = face['box']
            rois = [image_copy[y:y+h, x:x+w]]

            if face.size != 0:
                norm_face = mm.equalise_image(rois[0], eq_type='HSV')
                norm_face = mm.standardise_image(norm_face)
                norm_face = mm.resize_image(norm_face)

            if args.allModels or args.faceReco:
                probs_reco, y_hat_reco = mm.run_model(sess_reco, tf_placeholder_reco,   tf_cnn_softmax_reco, norm_face[None,:,:,:])

            if args.allModels or args.emotionDetect:
                probs_fer, y_hat_fer = mm.run_keras_model(model_fer, norm_face[None,:,:,:])

            if args.allModels or args.genderClass:
                probs_gender, y_hat_gender = mm.run_model(sess_gender, tf_placeholder_gender,  tf_cnn_softmax_gender, norm_face[None,:,:,:])

            # Define variables
            #myLabelText = num_to_class_gender[y_hat_gender[0]]+'-'+num_to_class_fer[y_hat_fer[0]]
            myLabelText = num_to_class_fer[y_hat_fer[0]]+' '+str(np.round(np.max(probs_fer, axis=1), 2)[0])
            myColor = bgr_colors[y_hat_reco[0]]
            myFontScale = (frame.shape[0] * frame.shape[1]) / (1000 * 1000)
            myFaceCenter = (int(x+w/2), int(y+h/2))
            myRadius = int(h/2)
            myCoordinates = x, y

            if y_hat_gender[0] == 0:
                rect = cv2.rectangle(frame, (x, y), (x+w, y+h), myColor, thickness=2)
            else:
                cirele = cv2.circle(frame, myFaceCenter, myRadius, myColor, thickness=2)

            # Tag
            cv2.putText(frame,
                        org=(myCoordinates),
                        text=myLabelText,
                        fontFace=fontFace,
                        fontScale=myFontScale,
                        color=green_color)

        # Display the resulting frame
        cv2.imshow('frame', frame)

        # Release keys
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # When everything done, release the capture
    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

D.10 Prepare Cropped Faces

```python
import matplotlib.pyplot as plt
import os
import numpy as np
import datetime
import cv2
import os

def convertToRGB(image):
    '''Convert image from BGR to RGB'''
    return cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

def save_detected_faces(cascade, image, scaleFactor = 1.1, output_dir = '/Users/Ekiavya/Movies/SortedFaces/2nd/', date_format = "%Y_%m_%d_%H_%M_%S_%f"):
    '''Saves the cropped faces in the given output directory'''
    roi_list = []
    image = convertToRGB(image)
    # Apply the haar classifier to detect faces
    # Run the cascade on the image, scaleFactor=scaleFactor, minNeighbors=5
    # For each face, get the region of interest and save it
    faces_rect = cascade.detectMultiScale(image, scaleFactor, minNeighbors=5)
    for i,(x, y, w, h) in enumerate(faces_rect):
        roi = image[y:y+h, x:x+w]
        cv2.imwrite(f'{output_dir}/{datetime.datetime.now().strftime(date_format)[:-4]}.png', convertToRGB(roi))

def get_images(folder, cascade):
    '''Returns images in directory path and returns the images's pixels as a list'''
    images = []
    for filename in sorted(os.listdir(folder)):
        # Read image
        img = cv2.imread(os.path.join(folder, filename))
        # Append
        if img is not None:
            save_detected_faces(cascade, img)

def main():
    haar_cascade_face = cv2.CascadeClassifier('data/haarcascades/haarcascade_frontalface_alt2.xml')
    images = get_images('/Users/Ekiavya/Movies/DadGoogliePhotos/', haar_cascade_face)

if __name__ == "__main__":
    main()
```

D.11 Convert Images to Data

```python
import pandas as pd
import numpy as np
import argparse
import cv2
import sys
import os

def convert_images_to_csv(args, img_size=20):
    '''Takes in directory path and returns the compressed images's pixels as a list'''

    images = []

    # Iterate through images in folder
    for i, filename in enumerate(sorted(os.listdir(args.input_dir))):
        if filename.split('.')[-1].lower() in {'jpeg', 'jpg', 'png'}:

            # Print progress
            if args.debug:
                percent = int((i+1)/len(os.listdir(args.input_dir))*100)
                if percent%10==0:
                    print(percent, '%')

            # Load and append
            img = cv2.imread(os.path.join(args.input_dir, filename))
            img = cv2.resize(img, (img_size, img_size))
            images.append(img)

    # Convert
    images = np.array(images)

    if args.debug:
        print('Saving as .csv file ...')

    # Save images as .csv
    np.savetxt(args.output_dir+'/'+args.filename+'.csv',
               images.reshape(images.shape[0], np.prod(images.shape[1:])),
               delimiter=',',
               fmt='%i')

def main():
    parser = argparse.ArgumentParser(description='Convert images in a folder to .csv')
    parser.add_argument('-d', '--debug', action='store_true', default=False, help='Print progress to stderr')
    parser.add_argument('-i', '--input_dir', type=str, action='store', help='Give input directory path')
    parser.add_argument('-o', '--output_dir', type=str, action='store', help='Give output directory path')
    parser.add_argument('-f', '--filename', type=str, default='image_data', action='store', help='Give output file name')
    args = parser.parse_args()

    images = convert_images_to_csv(args)

if __name__ == "__main__":
    main()
```

Appendix E

User Manual

GitHub Link

All files: https://github.com/EklavyaFCB/FacialInformationExtraction

Kaggle Link

Notebooks: https://www.kaggle.com/eklavyas/kernels
Personal dataset: https://www.kaggle.com/eklavyas/familyfaces
FER2013 dataset: http://tiny.cc/FER2013

Requirements

- `seaborn==0.9.0`
- `Keras==2.2.4`
- `numpy==1.16.4`
- `opencv_python_headless==4.1.0.25`
- `mtcnn==0.0.9`
- `albumentations==0.3.0`
- `matplotlib==3.1.0`
- `tensorflow==1.14.0`
- `pandas==0.24.2`
- `ipython==7.8.0`
- `MyMethods==1.0.0`
- `scikit_learn==0.21.3`

Installation

```
$ pip install -r requirements.txt
```

Sample Usage

```
$ python3 RestoreModel.py
$ python3 LiveInput.py
```

Common Errors

```
OpenCV(4.1.0) /Users/ travis/ build/ skvark/ opencv-python/ opencv/
modules/ objdetect/ src/ cascadedetect.cpp:1658: error: (-215:Assertion
failed) !empty() in function 'detectMultiScale'
```

A `.DS_Store` file in the directory `TestImages` containing the images. This can be solved by navigating to the directory containing the images, and delete the file in question.

```
$ cd TestImages
$ rm -rf .DS_Store
```

Deprecation Warnings
TensorFlow might give out a list of deprecation warnings - these can be safetly ignored.

Admin Permissions
Note that this script might require administrative permissions in order to use the device's webcam, which must often be password approved.

Appendix F

Miscellaneous

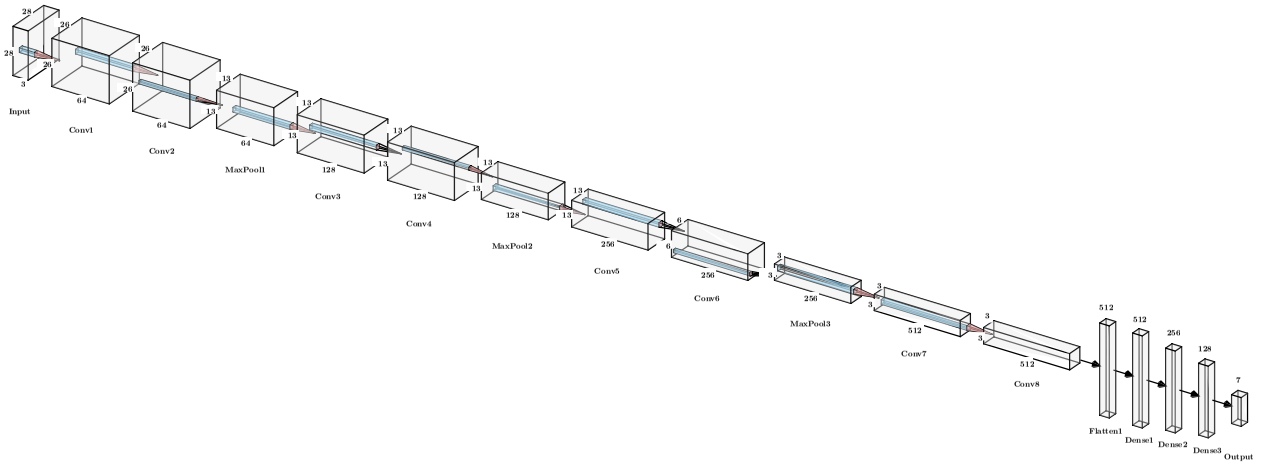

FIGURE F.1: Diagram of `MyDeepCNN` architecture.

Bibliography

Agui, T., Kokubo, Y., Nagashashi, H. and T, N., 1992. Extraction of face recognition from monochromatic photographs using neural networks. vol 1, p.pp 1881–1885.

Alom, M.Z., Taha, T.M., Yakopcic, C., Westberg, S., Hasan, M., Esesn, B.C.V., Awwal, A.A.S. and Asari, V.K., 2018. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164. 1803.01164, Available from: http://arxiv.org/abs/1803.01164.

Asthana, A., Zafeiriou, S., Cheng, S. and Pantic, M., 2014. Incremental face alignment in the wild. *2014 ieee conference on computer vision and pattern recognition*. pp.1859–1866. Available from: https://doi.org/10.1109/CVPR.2014.240.

Belhumeur, P.N., Hespanha, J.a. and Kriegman, D.J., 1996. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *Proceedings of the 4th european conference on computer vision-volume i - volume i*. Berlin, Heidelberg: Springer-Verlag, ECCV '96, pp.45–58. Available from: http://dl.acm.org/citation.cfm?id=645309.648965.

Benaich, N. and Hogarth, I., 2019. State of ai 2019. https://www.stateof.ai/. (Accessed on 09/10/2019).

Bloice, M.D., Roth, P.M. and Holzinger, A., 2019. Biomedical image augmentation using Augmentor. *Bioinformatics*. Available from: https://doi.org/10.1093/bioinformatics/btz259.

Burel, G. and Carel, D., 1994. Detection and localization of faces on digital images. *Pattern Recognition Letters*, 15, pp.963–967. Available from: https://doi.org/10.1016/0167-8655(94)90027-2.

Buslaev, A.V., Parinov, A., Khvedchenya, E., Iglovikov, V.I. and Kalinin, A.A., 2018. Albumentations: fast and flexible image augmentations. *CoRR*, abs/1809.06839. 1809.06839, Available from: http://arxiv.org/abs/1809.06839.

Chauvet, J., 2018. The 30-year cycle in the AI debate. *CoRR*, abs/1810.04053. 1810.04053, Available from: http://arxiv.org/abs/1810.04053.

Chollet, F., 2017. *Deep learning with python*. 1st ed. Greenwich, CT, USA: Manning Publications Co.

Cootes, T., Edwards, G. and Taylor, C., 2001. Active appearance models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23, pp.681 – 685. Available from: https://doi.org/10.1109/34.927467.

Cutting edge tensorflow - keras tuner: hypertuning for humans - io 2019. https://elie.net/talk/cutting-edge-tensorflow-keras-tuner-hypertuning-for-humans/. (Accessed on 09/03/2019).

Farfade, S.S., Saberian, M.J. and Li, L., 2015. Multi-view face detection using deep convolutional neural networks. *CoRR*, abs/1502.02766. 1502.02766, Available from: http://arxiv.org/abs/1502.02766.

Feraud, R., Bernier, O.J., Viallet, J.E. and Collobert, M., 2001. A fast and accurate face detector based on neural networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(1), pp.42–53. Available from: https://doi.org/10.1109/34.899945.

Goodfellow, I. et al., 2013. Challenges in representation learning: A report on three machine learning contests. Available from: http://arxiv.org/abs/1307.0414.

Guo, G., Wang, H., Yan, Y., Zheng, J. and Li, B., 2018. A fast face detection method via convolutional neural network. *CoRR*, abs/1803.10103. 1803.10103, Available from: http://arxiv.org/abs/1803.10103.

Jabid, T., Hasanul Kabir, M. and Chae, O., 2010. Local directional pattern (ldp) for face recognition. vol. 8, pp.329 – 330. Available from: https://doi.org/10.1109/ICCE.2010.5418801.

James, G., Witten, D., Hastie, T. and Tibshirani, R., 2014. *An introduction to statistical learning: With applications in r*. Springer Publishing Company, Incorporated.

Jung, A.B., 2018. imgaug. https://github.com/aleju/imgaug.

keras-team/keras-tuner: Hyperparameter tuning for humans. https://github.com/keras-team/keras-tuner. (Accessed on 09/03/2019).

Khorrami, P., Paine, T.L. and Huang, T.S., 2015. Do deep neural networks learn facial action units when doing expression recognition? *CoRR*, abs/1510.02969. 1510.02969, Available from: http://arxiv.org/abs/1510.02969.

Koch, G., Zemel, R. and Salakhutdinov, R., 2017. Siamese neural networks for one-shot image recognition. Available from: http://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf.

Kohonen, T., 1989. *Self-organization and associative memory: 3rd edition*. Berlin, Heidelberg: Springer-Verlag.

Krishna, R., 2018. Stanford university cs131: Foundations and applications of computer vision. http://vision.stanford.edu/teaching/cs131_fall1819/index.html. (Accessed on 03/06/2019).

Kumar, A., Kaur, A. and Kumar, M., 2018. Face detection techniques: a review. *Artificial Intelligence Review*. Available from: https://doi.org/10.1007/s10462-018-9650-2.

Kumari, J., Rajesh, R. and Pooja, K., 2015. Facial expression recognition: A survey. *Procedia Computer Science*, 58, pp.486 – 491. Second International Symposium on Computer Vision and the Internet (VisionNet'15). Available from: https://doi.org/https://doi.org/10.1016/j.procs.2015.08.011.

LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D., 1989. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), pp.541–551. Available from: https://doi.org/10.1162/neco.1989.1.4.541.

Li, F.F. and Karpathy, A., 2018. Stanford university cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/. (Accessed on 03/06/2019).

Lin, S.H., Kung, S.Y. and Lin, L.J., 1997. Face recognition/detection by probabilistic decision-based neural network. *Trans. Neur. Netw.*, 8(1), pp.114–132. Available from: https://doi.org/10.1109/72.554196.

Mayya, V., Pai, R.M. and Pai, M.M., 2016. Automatic facial expression recognition using dcnn. *Procedia Computer Science*, 93, pp.453 – 461. Proceedings of the 6th International Conference on Advances in Computing and Communications. Available from: https://doi.org/https://doi.org/10.1016/j.procs.2016.07.233.

McCloud, S., 2006. *Making comics : storytelling secrets of comics, manga and graphic novels*. 1st ed. New York : Harper. Available from: https://search.library.wisc.edu/catalog/9910022219902121.

Megagenta, Z. and Sarkar, E., 2019. *Machine learning 2: Neural networks coursework*. University of Bath. Available from: https://www.dropbox.com/s/b5jh9kqfa0p09z9/Lab_2___Neural_Networks.pdf?raw=1.

Mikołajczyk, A. and Grochowski, M., 2018. Data augmentation for improving deep learning in image classification problem. *2018 international interdisciplinary phd workshop (iiphdw)*. pp.117–122. Available from: https://doi.org/10.1109/IIPHDW.2018.8388338.

Minar, M.R. and Naher, J., 2018. Recent advances in deep learning: An overview. *CoRR*, abs/1807.08169. 1807.08169, Available from: http://arxiv.org/abs/1807.08169.

Mozur, P. and Lin, Q., 2019. In hong kong protests, faces become weapons - the new york times. https://www.nytimes.com/2019/07/26/technology/hong-kong-protests-facial-recognition-surveillance.html. (Accessed on 09/10/2019).

Oh, K.S. and Jung, K., 2004. Gpu implementation of neural networks. *Pattern Recognition*, 37(6), pp.1311 – 1314. Available from: https://doi.org/https://doi.org/10.1016/j.patcog.2004.01.013.

Opencv: Face detection using haar cascades. https://docs.opencv.org/3.3.0/d7/d8b/tutorial_py_face_detection.html. (Accessed on 03/06/2019).

Osuna, E., Freund, R. and Girosit, F., 1997. Training support vector machines: an application to face detection. *Proceedings of ieee computer society conference on computer vision and pattern recognition*. pp.130–136. Available from: https://doi.org/10.1109/CVPR.1997.609310.

Ou, J., 2012. Classification algorithms research on facial expression recognition. *Physics Procedia*, 25, pp.1241 – 1244. International Conference on Solid State Devices and Materials Science, April 1-2, 2012, Macao. Available from: https://doi.org/https://doi.org/10.1016/j.phpro.2012.03.227.

Perez, L. and Wang, J., 2017. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621. 1712.04621, Available from: http://arxiv.org/abs/1712.04621.

Phillips, P.J. et al., 2018. Face recognition accuracy of forensic examiners, superrecognizers, and face recognition algorithms. *Proceedings of the National Academy of Sciences*, 115, p.201721355. Available from: https://doi.org/10.1073/pnas.1721355115.

Pramerdorfer, C. and Kampel, M., 2016. Facial expression recognition using convolutional neural networks: State of the art. *CoRR*, abs/1612.02903. 1612.02903, Available from: http://arxiv.org/abs/1612.02903.

Propp, M., Samal and Ashok, K., 1992. Artificial neural network architectures for human face detection. In: C. Dagli, L. Burke and Y. Shin, eds. *Intelligent engineering systems through artificial neural networks*. ASME, vol. 2, pp.535–540.

Rowley, H., Baluja, S. and Kanade, T., 1996. Neural network-based face detection. vol. 20, pp.203–208. Available from: https://doi.org/10.1109/34.655647.

Saez-Trigueros, D., Meng, L. and Hartnett, M., 2018. Face recognition: From traditional to deep learning methods. *CoRR*, abs/1811.00116. 1811.00116, Available from: http://arxiv.org/abs/1811.00116.

Sarkar, E., 2018. Artificial neural networks: Kohonen self-organising maps. https://eklavyafcb.github.io/som.html. (Accessed on 03/06/2019).

Schroff, F., Kalenichenko, D. and Philbin, J., 2015. Facenet: A unified embedding for face recognition and clustering. *2015 ieee conference on computer vision and pattern recognition (cvpr)*. pp.815–823. Available from: https://doi.org/10.1109/CVPR.2015.7298682.

Taigman, Y., Yang, M., Ranzato, M. and Wolf, L., 2014a. Deepface: Closing the gap to human-level performance in face verification. *2014 ieee conference on computer vision and pattern recognition*. pp.1701–1708. Available from: https://doi.org/10.1109/CVPR.2014.220.

Taigman, Y., Yang, M., Ranzato, M. and Wolf, L., 2014b. Deepface: Closing the gap to human-level performance in face verification. *2014 ieee conference on computer vision and pattern recognition*. pp.1701–1708. Available from: https://doi.org/10.1109/CVPR.2014.220.

Tarnowski, P., Kołodziej, M., Majkowski, A. and Rak, R.J., 2017. Emotion recognition using facial expressions. *Procedia Computer Science*, 108, pp.1175 – 1184. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. Available from: https://doi.org/https://doi.org/10.1016/j.procs.2017.05.025.

Turk, M. and Pentland, A., 1991. Eigenfaces for recognition. *J. Cognitive Neuroscience*, 3(1), pp.71–86. Available from: https://doi.org/10.1162/jocn.1991.3.1.71.

UK, G., 2019. Ethical issues arising from the police use of live facial recognition technology. https://www.gov.uk/government/publications/police-use-of-live-facial-recognition-technology-ethical-issues. (Accessed on 09/10/2019).

Viola, P. and Jones, M., 2001. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 ieee computer society conference on computer vision and pattern recognition. cvpr 2001.* vol. 1.

Viola, P. and Jones, M., 2001. Rapid object detection using a boosted cascade of simple features. vol. 1, pp.I–511. Available from: https://doi.org/10.1109/CVPR.2001.990517.

Wikipedia, 2019. Histogram equalization. https://en.wikipedia.org/wiki/Histogram_equalization. (Accessed on 08/14/2019).

Winston, A., Beachy, S., Bennett, K. and Delaquérière., A., 2019. Facial recognition: Dawn of dystopia, or just the new fingerprint? - the new york times. https://www.nytimes.com/2019/05/18/us/facial-recognition-police.html. (Accessed on 09/10/2019).

Wong, K.W., Lam, K.M. and Siu, W.C., 2001. An efficient algorithm for human face detection and facial feature extraction under different conditions. *Pattern Recognition - PR*, 34, pp.1993–2004. Available from: https://doi.org/10.1016/S0031-3203(00)00134-5.

Woodhouse, J., Pratt, A., Brown, J., Hutton, G. and Wilkins, H., 2019. Facial recognition and the biometrics strategy. https://researchbriefings.parliament.uk/ResearchBriefing/Summary/CDP-2019-0099. (Accessed on 09/10/2019).

Wu, Y. and Ji, Q., 2018. Facial landmark detection: a literature survey. *CoRR*, abs/1805.05563. 1805.05563, Available from: http://arxiv.org/abs/1805.05563.

Yang, G. and Huang, T.S., 1994. Human face detection in a complex background. *Pattern Recognition*, 27(1), pp.53 – 63. Available from: https://doi.org/https://doi.org/10.1016/0031-3203(94)90017-5.

Yang, M.H., Kriegman, D.J. and Ahuja, N., 2002. Detecting faces in images: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(1), pp.34–58. Available from: https://doi.org/10.1109/34.982883.

Zhang, Z., Luo, P., Loy, C.C. and Tang, X., 2015. Learning social relation traits from face images. *CoRR*, abs/1509.03936. 1509.03936, Available from: http://arxiv.org/abs/1509.03936.