Artificial Intelligence Techniques for Driver Fatigue Detection

by

Nabil Yassine

OXFORD
BROOKES
UNIVERSITY

A thesis submitted in partial fulfilment of the requirements
of Oxford Brookes University for the degree of
Doctor of Philosophy

June 2020

ACKNOWLEDGMENTS

I would like to thank my supervisors Prof. Khaled Hayatleh, Dr Steve Barker and Prof. Bhaskar Choubey, for their support and guidance throughout my PhD research.

I would also like to thank Philip Hughes, Saddam Zourob, Prof. John Durodola and Rajasekhar Nagulapalli for their moral and technical support throughout my research.

Lastly, I would like to my parents, my siblings and my extended family for their support and encouragement during my studies.

Nabil Yassine
Oxford Brookes University
Oxford
August 2020

ABSTRACT

The research discussed here aims to design a deep learning algorithm based on Convolutional Neural Networks models to detect driver distraction and fatigue using driver facial expressions. The proposed model provides high accuracy during both training and validation. The research was inspired to contribute to transport safety by providing alternative solutions to detect driver habit.

First, the thesis discussed Conventional methods, including Haar cascade classifiers and eigenfaces. In 2018 I published a proposal for a blink rate detection system using Haar cascade feature detection. However, due to the advantages of Neural Networks, the research focused on providing a unique solution in that field. An in-depth look at how Neural Networks function, specifically Convolutional Neural Networks (CNNs), was investigated and discussed next. Due to the advantages CNN's have with feature detection in images, the algorithm I proposed in this research uses a CNN architecture. Lastly, I proposed an adaptive approach for deep learning to enhance training, validation and testing accuracies.

My original algorithm and subsequent models were trained on two datasets. These were the American University in Cairo (AUC) Distracted Driver Dataset and the UTA Real-Life Drowsiness Dataset (UTA-RLDD). Hence the research proposed two original CNN models that produced high training and validation accuracy. The model designed on the AUC Distracted Driver Dataset achieved good 97% training accuracy and good 96% validation accuracy. Evaluation of this model produced good 99% accuracy. The model designed on UTA-RLDD achieved 100% training accuracy, 69% validation accuracy, and evaluated at 69% accuracy.

LIST OF ACRONYMS

AdaBoost	a method used to remove irrelevant features
Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
ANN	Adaptive Neural Network
API	Application Programming Interface
ATCs	Air traffic controllers
AUC	American University in Cairo
CNN	Convolutional Neural Network
DaCoTa	a project co-financed by the EU mobility and Transport
Eigenfaces	an appearance-based approach to facial recognition
ISP	image signal processing
mpl	matplotlib
Nadam	Nesterov Accelerated Adam Nadam
NAG	Nesterov accelerated gradient
NN	Neural Network
np	numpy
OpenCV	an open source computer vision library
pd	pandas
plt	matplotlib.pyplot
RBM	Restricted Boltzmann machine networks

ROSPA	Royal Society for the Prevention of Accidents
SGD	Stochastic Gradient Descent
tf	tensorflow
TOT	Time on task
UTA-RLDD	University of Texas at Arlington Real-Life Drowsiness Dataset

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

1.1 Rationale

Drivers suffer from fatigue resulting in thousands of road accidents each year. According to the Royal Society for the Prevention of Accidents (ROSPA), research has shown that driver fatigue may contribute to road accidents by up to 20%. Also, driver fatigue contributes to up to 25% of severe and fatal accidents [1]. Drivers with high fatigue levels may fall asleep while driving, or have slow response times when trying to brake. Hence this increases the likely hood of death or serious injury occurring by up to 50% [1]. Hunter [2] mentioned a survey in 2012 that 29% of drivers admitted having almost fallen asleep at the wheel. The fatigued driver is not only putting themselves at risk but also other drivers, passengers and pedestrians. Research had shown [2] that performance levels when under the influence of alcohol or sleep deprivation showed similar degradations [2].

DaCoTa, a project co-financed by the EU mobility and Transport [3], described slower reaction times, difficulty in estimation of speeds and headways, reduced attention to road signs and crossing a traffic light without noticing its colour as clear examples of reduced levels of concentration and awareness due to fatigue. In terms of Driving, these are all observations of how well the driver is driving. A system looking at these would need the levels of fatigue to reach a point when the driving is being affected, at which point the system would start to evaluate. However, developing a system which observed the driver themselves could inform them of their fatigue levels earlier on.

Figure 1.1 shows how fatigue leads to a reduced driving performance resulting in slower reaction time, reduced steering performance and increased distraction from being aware of nearby surroundings [3]. The withdrawal of attention and cognitive processing capacity from the driving task is a subconscious reaction; hence the driver would be initially unaware of being tired [3]. However, once they become conscious of it, drivers may try to change their driving habit, once they realise they are tired [3]. For example, they may drive faster so that a ‘new’ sensation of driving raises adrenaline and attention levels, or they may slow down to increase the distance between the car in front [3]. However, crashes and observations of the driving performance show that compensatory strategies are not sufficient to remove all excess risk [3]. Figure 1.1 also shows how driver fatigue has an impact on driving performance, including the driver’s task motivation in maintaining safe

driving techniques. Fatigue levels not only reduce awareness levels but can also lead to the driver being distracted and performing tasks other than focusing on driving [3, 4].

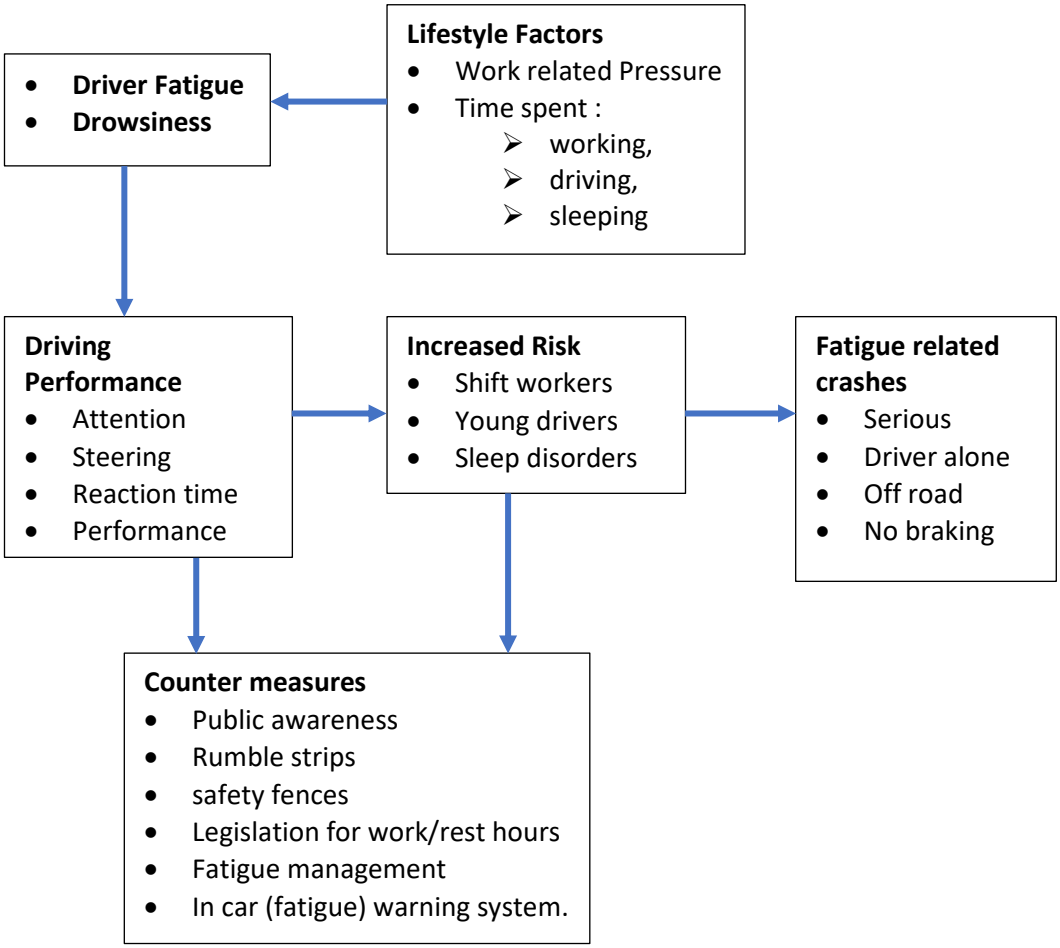

Fig. 1.1 The effects of fatigue [3].

According to ROSPA [1] crashes caused by drivers occur mostly when on long journeys, in the early hours of the morning (2 am to 6 am), between 2 pm and 4 pm after a heavy meal, drinking, or driving back from work, the driver being sleep deprived and taking medication. To reduce the risk and fatigue levels while driving ROSPA recommends pre-planning your journey with rests and having a good night's sleep beforehand.

ROSPA also suggests using fatigue detection and warning systems, but at the same time, warns that drivers may end up relying on them when they feel tired. Hence, they may drive while feeling tired and hope the system informs them in time

[1]. Developing a system that warns a driver of increased fatigue levels allows them to take appropriate action, thereby reducing the risk of a collision and injury to themselves and any other potential victim(s). However, it will not necessarily deal with the concerns claimed by ROSPA.

Other issues vary with differing applications. A review by Stern, Boyer and Schroeder [5] showed that fatigue has a significant impact on the ability of pilots and air traffic controllers (ATCs) to maintain their performance with time [5]. They suggest that Blink rate can be one of several psychophysiological measures to determine fatigue levels in individuals [5]. The review provided evidence that other factors, other than time on task (TOT) performance, affected the blink rate just as much [5]. These additional factors indicate that a fatigue detection system would have multiple uses, not just in the aviation and motoring industries.

Regarding the aviation industry, fatigue levels of airline pilots can be subjective. When asking a pilot to describe their awareness level, they may not feel tired, but when testing their reaction times, may show they are tired [2]. However, when confronting the pilot with a test, they can end up becoming more alert which would provide inaccurate results [2].

1.2 Benefits of Fatigue Detection Systems

A measurement system providing warning systems for driver fatigue would have a significant impact on reducing the likelihood of accidents [3]. By reducing the time, it takes for the driver to be consciously aware of the drop-in attention, due to fatigue, their reaction times will not delay as much and should reduce the chance of an accident occurring [6].

Fatigue monitoring systems have the potential to be used in any situation where fatigue causes an increased risk of safety, not just driving. Security personnel working on x-ray machines at an airport and airline pilots are examples of other professionals that could benefit from this system. The aviation industry has noted that some notable airline accidents were due to pilot error, caused by the direct influence of pilot fatigue [2], thereby demonstrating the safety benefits that this system would bring.

The main interest in fatigue awareness systems has come from the motor industry. Car manufactures such as BMW and Ford have developed their own awareness systems. Their respective patents are critically analysed in the literature review (chapter two). Other methods, such as Optalert glasses [7, 8] that measure the blink rate of drivers to determine fatigue levels, have also been looked investigated. A specification was drawn up from this analysis to develop a system that would provide an original contribution to this field.

1.3 Aim and Objectives

This research aimed to develop a machine-learning algorithm to detect fatigue and driver distraction. The system was designed to detect fatigue using facial expression as the main indicator, as well as driver distraction.

To achieve the aim, the following objectives needed to be met.

- Review the existing literature to investigate how facial expression and driver distraction are related to fatigue levels – and hence performance levels.
- Review existing awareness/fatigue and driver distraction technologies.
- Develop high accuracy (at least 90%) of artificial intelligence-based techniques for fatigue and distraction detection
- Train the AI system using a database of images of human faces.
- Evaluate and refine the training algorithm.
- Test and optimise the AI system.
- Analyse and discuss the results.

1.4 Original Contribution

The original contribution, presented in this thesis, was to develop an improved AI-based technique that can identify fatigue by observing the facial expressions of participants in databases of human faces. The dataset used to classify fatigue was the University of Texas at Arlington Real-Life Drowsiness Dataset (UTA-RLDD) [9]. The proposed Neural Network model was also able to identify specific actions considered distractive behaviour when driving. The dataset used to classify these distractive behaviours was the American University in Cairo (AUC) Distracted Driver Dataset [10, 11]. The model trained on the dataset produced a good accuracy of 99%, while a similar model trained and validated on the fatigued dataset produced a good accuracy of 69%. The research investigations, presented in this thesis, aimed to introduce novel features of adaptive machine learning by observing the effects specific parameters have on the accuracy of Neural Networks during training and validation phases and using these observations during the real-time learning process.

1.5 Considerations and Specifications

When developing a Neural Network system to detect fatigue and distraction, it is imperative that the system is trained on a wide range and plenty of relevant samples. These samples should increase the likelihood the system is tested on its intelligence, rather than its memory. The primary condition to ensure the Neural Network being tested on its intelligence is for its testing data to be a wide range and never been presented to the Neural Network before.

An important consideration to make is looking at the blink rate as a measurement of fatigue. In normal rest condition, a human's average blink rate is around 17 blinks per minute [9]. Depending on a specific activity, the blink rate can go as high as 26 blinks per minute, to as low as 4-5 blinks per minute [9]. While driving the blink rate can be less than in resting conditions, at around 4-5 blinks per minute [9]. Conventional techniques (discussed in chapter two) use blink rate as a measurement of fatigue. The research presented in this thesis tackles fatigue detection by looking at changes to facial expression using a Neural Network approach.

Finally, five main factors that cause fatigue are a lack of energy, physical exertion, physical discomfort, lack of motivation and sleepiness. These factors can increase the chances the individual becomes distracted and perform tasks to increase their awareness levels [4]. It would be useful to consider that a fatigue awareness system is also capable of identifying actions taken by participants due to the link between fatigue and distraction [4].

1.6 Structure of the Thesis

This thesis is divided into nine chapters. To make the reading of this thesis easier, I discuss the major points of the algorithm, with the code repeated in full in the appendices. Each chapter has their own reference section, where appropriate, as well as a summary at the end of each chapter.

Chapter one covers the rationale of the project and sets out the aim and objectives. Chapter two is a review of the literature which examines and critically evaluates the current technologies related to this research. These technologies include blink rate detection systems, such as the Optalert alertness sensing device. Other Devices and systems from mainstream car manufacturers, such as BMW and Ford, are critically analysed as well. One of the critical aspects of the proposed design in this thesis is image processing and convolution techniques. Also examined and critically evaluated are Patents related to the field.

Chapter three discusses the principles behind Neural Network models and how they are built and function. The chapter focuses on convolutional Neural Networks which are the key features of the proposed Neural Network model presented in this work. The disadvantages and advantages of these models are also discussed.

Chapter four discusses the proposed system. This involves looking at how the Neural Network system achieves its goal and discusses why it was designed in this way.

Chapter five evaluates the final product design explained in chapter four. This covers the extensive testing and evaluation of the AI system. Following this testing, the system is optimised and re-tested to confirm conformity to the original design specification. The results of the testing are then discussed.

Chapter six is the conclusion of the work and discusses the outcomes of the research as a whole. The advantages and limitations are presented, alongside recommendations for future work.

Chapters seven is a full reference list (in alphabetical order) for all the chapters in this thesis.

Chapter eight is the appendices. This includes program code listings, and supplementary material relating to the previous chapters.

Chapter nine includes papers published by the author of this thesis during the research.

1.7 Summary

This chapter presented the rationale for the work. It then continued to cover the aim and the objectives needed to achieve this aim. The original contribution of the approaches was also stated. Finally, the thesis structure was outlined.

The following chapter is the literature review. This chapter includes a full critical evaluation and discussion of contemporary approaches and technologies. Lastly, there is an evaluation of related Patents.

1.8 References

[1] ROSPA. "Driver Fatigue and Road Accidents Factsheet." https://www.rospa.com/media/documents/road-safety/driver-fatigue-factsheet.pdf (accessed February, 2020).

[2] R. Hunter, "Staying Awake, Staying Alive: The problem of fatigue in the transport sector," PACTS, London, 2013.

[3] "DaCoTA (2012) Fatigue," ed. Deliverable 4.8h of the EC FP7 Project DaCoTA.

[4] A. Williamson, "21 The Relationship between Driver Fatigue and Driver Distraction," *Driver distraction: Theory, effects, and mitigation,* p. 383, 2008.

[5] J. A. Stern, D. Boyer, and D. J. Schroeder, "Blink Rate As a Measure of Fatigue: A Review," Department of Psychology, Washington University, National Technical Information Service, 1994.

[6] W. Tansakul and P. Tangamchit, "Fatigue Driver Detection System Using a Combination of Blinking Rate and Driving Inactivity," *Journal of Automation and Control Engineering,* vol. 4, no. 1, 2016.

[7] "Optalert Adopts InterSystems Software to Prevent Fatigue-Related Industrial Accidents," ed. Sydney: InterSystms Cache and DeepSee Case Study, 2010.

[8] R. Murray Johns and R. Christopher Hocking, "Alertness sensing device," United States of America Patent 9,007,220, 2015.

[9] R. Ghoddoosian, M. Galib, and V. Athitsos, "A Realistic Dataset and Baseline Temporal Model for Early Drowsiness Detection," 2019, pp. 0-0.

[10] Y. Abouelnaga, H. M. Eraqi, and M. N. Moustafa, "Real-time distracted driver posture classification," *arXiv preprint arXiv:1706.09498,* 2017.

[11] H. M. Eraqi, Y. Abouelnaga, M. H. Saad, and M. N. Moustafa, "Driver distraction identification with an ensemble of Convolutional Neural Networks," *Journal of Advanced Transportation,* vol. 2019, 2019.

Chapter 2: Literature Review

2.1 Introduction

With car safety regulations consistently being revised and updated over time, developing a system that looks explicitly at drowsiness, and how this affects driving safety, has become more relevant for major car manufactures [12]. Currently developed systems looking into fatigue detection/monitoring follow a general pattern. These systems include monitoring the driver's awareness and setting off an alarm when the driver is perceived to be suffering from a lack of awareness, such as fatigue or being distracted. The monitoring of the driver can include steering wheel monitoring, lane deviation and distracted behaviour [12, 13].

Two commonly used detection methods are Haar cascade classifiers, developed by Viola and Jones [14], and Eigenfaces, used for face classification by Turk and Pentland [15]. Haar cascade classifiers work by identifying features of the face, such as eyes and mouth. Eigenface techniques look at the face as a whole to identify faces in images.

Fatigue detection systems require constant observation of the user's face or lane deviation systems to determine the state of the user or their driving technique. Lane deviation systems use car-mounted cameras that detect road markings and use their location to establish the car's position. Hence, the cameras used for this are exteriorly mounted or exterior looking sensors. These systems check to see if the car leaves the lane between those road markings without a clear purpose, for instance, where there is no use of the indicator. Finally, the system would then warn the driver [12]. Monitoring the user's face uses a similar technique, except the cameras would be inside the vehicle pointed at the drivers face. These systems are typically designed to detect a change, such as the blink rate or facial expression, to warn the driver. Both systems require a specified threshold that would need to be met.

This chapter discusses the two mentioned detection methods that can be used to detect the face of an individual. Lastly, a list of patents on systems that monitor driver behaviour and technique are also discussed.

2.2 Haar Cascade Classifiers

Various detection methods were considered, including the method proposed by Lalonde *et al*. [16]. This involved simple motion detection, based on looking at the difference between frames inside the tracked regions of the eye [16]. An approach that is commonly used involves Haar-cascade classifiers [17-19], which was initially proposed by Viola and Jones [14].

Viola and Jones [14] proposal was based on the use of machine learning for visual object detection capable of highly successful image detection at high rates [14]. The detection system they proposed had three key features. The first is the integral image which needs high computation rates for the features used by their detector [14]. The second was their learning algorithm based on AdaBoost [20], which is a method used to remove irrelevant features. This effectively improved their learning algorithm by selecting specific features from a large data pool and providing efficient classifiers [20]. The third was combining the classifiers into a cascade which allowed for much faster processing by discarding the images' background regions, hence eliminating the need to scan these areas [20]. Drawing from this method, the first stage of the algorithm is to train the classifier using positive (images containing faces) and negative images (without faces) [17-19]. The next stage is to identify features from the images.

2.2.1 Feature Identification

Figures 2.1, 2.2 and 2.3 show the Haar features used. Note the Haar features are very similar to convolutional kernels (a small matrix used for blurring, sharpening images as well as for edge detection). Each feature is a single value obtained by subtracting the sum of pixels under white rectangles from the sum of pixels under black rectangles [17-19]. Edge features (figure 2.1) are used to detect edges in images, where there is an apparent transition between pixel values. These are good at detecting eyebrows. Line features (figure 2.2) are used to detect features that are surrounded by pixels of a different value (intensity) compared to the pixels that make up your feature. Lips are commonly detected using line features. Finally, the Four rectangle features (figure 2.3) calculates the difference between opposite pairs of rectangles [17].

Fig. 2.1 Common Haar edge feature [17].

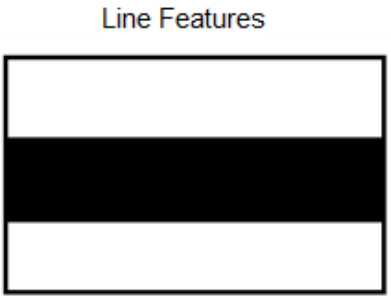

Fig. 2.2 Common Haar line feature [17].

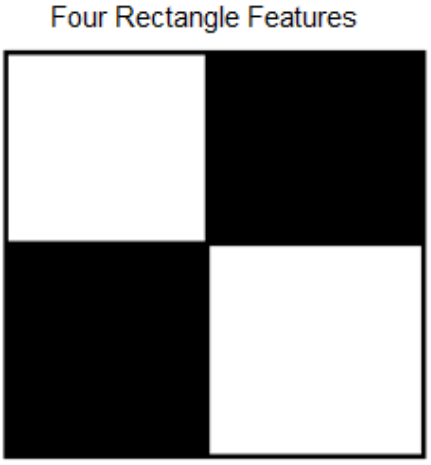

Fig. 2.3 Common Haar four rectangle feature [17].

Looking at figure 2.4, the leftmost image is the original image [17]. The top centre image shows a common Haar edge feature (figure 2.1) being used, while the top

right image is using a common Haar line feature. The bottom two images show the detection of the eyes and nose in the image (without the original image as the background). The common Haar edge feature (example can be seen in figure 2.1) is used to detect the distance between the eyes (black part of the horizontal rectangle) and the distance between the nose and cheeks (the white part of the horizontal rectangle). The distinction is shown due to the eyes being darker than the nose and cheeks. Similarly, the two most right-hand images, using the common Haar line feature (example can be seen in figure 2.2), show the distinction between the eyes (black part of the rectangle) and the bridge of the nose (white part of the rectangle) [18, 19]. Again, this is achieved because the eyes are considered darker than the bridge of the nose.

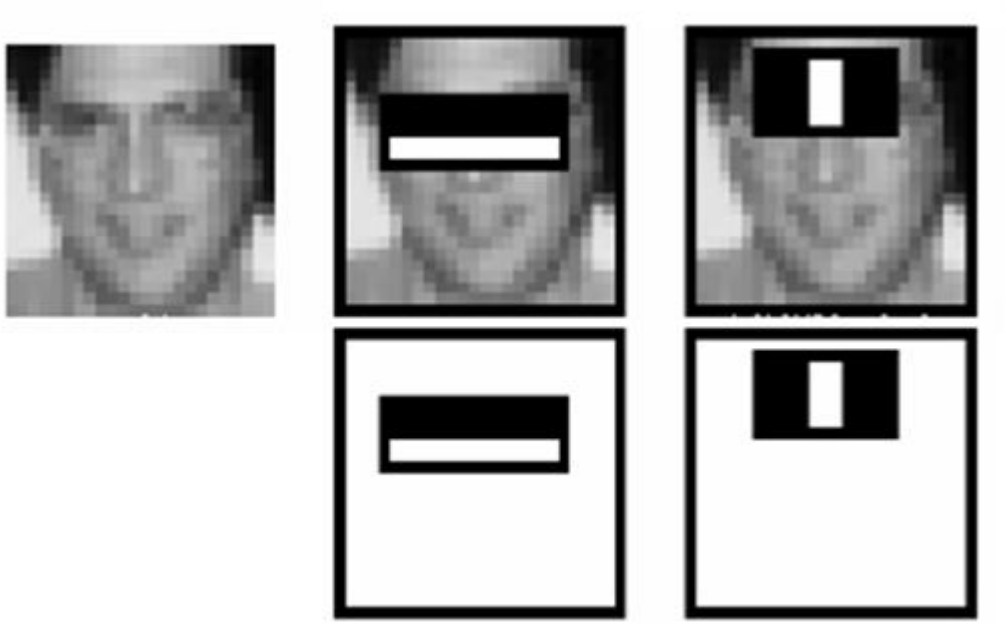

Fig. 2.4 A mixture of edge features and line features used to detect facial features [17].

Since these are greyscaled images, each pixel value can vary from 0 to 255 [17]. If the white pixels are assigned as 255 and black pixels as 0, it can be deduced that the white parts (cheeks and nose) were created using pixel values close to 255. Summing up these values would give a much higher total value compared to the eye regions, which are darker regions and so have a much smaller total pixel value. Hence the higher total pixel value is shown as lighter (with 255 being white), and the lower pixel value is shown as darker (with 0 being black) [17].

2.2.2 Integral Image

Processing an image can become very inefficient and time-consuming, especially as the size of the image increases. When calculating each feature, it would be necessary to find the sum of pixels under the white part of the rectangle and the black rectangle. However, this can get out of hand because it would be required to calculate all possible sizes of features.

The Haar cascade provided by OpenCV [17, 21], solves this issue by using the integral image [17] instead of the original image (figure 2.5). The integral image is calculated by summing up the pixel values on an image in the area above and to the left of the pixel location. Looking at figure 2.5, the original image is first padded with zeros above and to the left. At each pixel location, the value is replaced with the sum of all the values above and to the left of the pixel location, in the original image. Figure 2.5 shows the value of 6 circled in the integral image. It is calculated from the sum of values circled in the original image padded with zeros, which is done for each pixel location. The integral image can be easily used to determine what each region of the image is. As previously stated, the higher the pixel value, the lighter the pixel and the smaller the value, the darker the pixel. So, darker regions of the image have smaller values in the integral image and lighter regions have larger values.

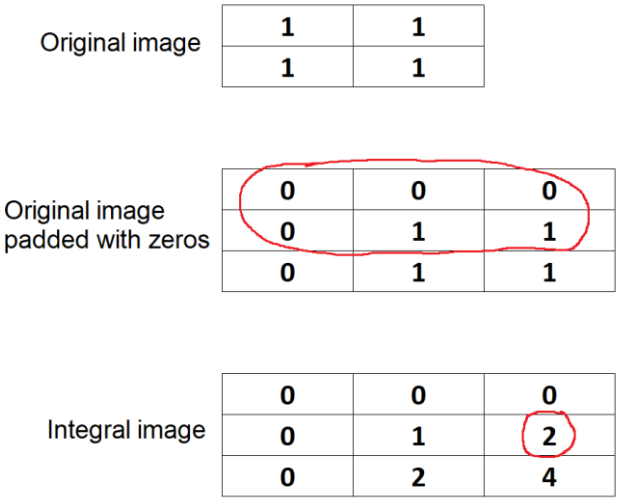

Fig. 2.5 Example of a resultant integral image calculated from the original padded image.

The significant advantages of this are reduced computational time and power requirements. While the underlining issue with this method is that the calculations can be described as guesswork, many of the features detected are not relevant when carrying out the image processing [17].

Tansakul *et al*., [6] used Haar-cascade classifiers for face and eye detection. They first detected the face, then once in the face region of the image, searched for the eyes. However, they ran into issues using Haar-cascade, due to it being computationally expensive, giving slow refresh rates when compared to the speed of blink rates. To solve this issue, they extracted the eye region of the face by applying template matching [6]. This reduced the size of the image the tracking would need to take place on, thereby reducing the computational time that Haar-cascade classifier would typically require.

Another method to remove the irrelevant features, a technique called AdaBoost was developed by Freund and Schapire [22], which was based on the Vapnik–Chervonenkis theory [23]. AdaBoost is an algorithm technique that improves the accuracy of a learning algorithm. All analyses of learning methods depend in some way on assumptions, since otherwise, learning is quite impossible. These assumptions are based on the hypotheses presented to have improved accuracy and efficiency compared to random guessing. Its development solved practical limitations and difficulties which were found in earlier boosting algorithms.

2.3 Eigenfaces

Eigenfaces is an appearance-based approach to facial recognition, compared to feature-based (object) detection approaches, such as the Haar cascade approach [18, 24]. The approach involves comparing face images in their entirety rather than looking for features and comparing them, for example, checking for shape and colour of the eyes.

The idea behind eigenface was to base face recognition on small sets of image features. These features would best approximate the set of known face images, without being dependent on specific facial parts and features that would be considered intuitive, such as the nose or eyes.

The first step is to obtain a training dataset of face images, which must be centred and have equal image dimensions. Considering that each pixel in an image represents a number, matrix I_j can be used to represent the image with dimensions NxN. Note j represents a number between 1 and m, where m is the total number of images (figure 2.6). The 2-dimensional matrices I_i are then flattened into one-dimensional vectors, u_i [15, 25].

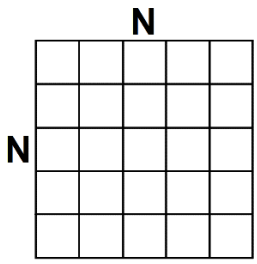

Fig. 2.6 Matrix I_j with dimensions NxN [15].

Next, the average face vector (μ) is calculated by the sum of each (vector μ_i) divided by the total number of images (M) (see equation 2.1). Once calculated, the mean face is subtracted from each image in the image vector μ_i (see equation 2.2). Hence, this produces a new vector, Φ_i(phi)[15, 25].

$$\mu = \frac{1}{M}\sum_{i=1}^{M} u_i \qquad \text{(Eq 2.1)}$$

$$\Phi_i = u_{\mathrm{i}} - \mu \qquad \text{(Eq2.2)}$$

The covariance matrix (C) can now be calculated by grouping the vectors u_n, which represents all Φ_i(n ranges from 1 to M, where M is the total number of images). Hence, this forms a new matrix called A. Covariance, C, (figure 2.7) is the multiplication of the new matrix A with its transposition, called A^T. Looking at figure 2.7, each row for matrix A and each column of A^T represents a vector, Φ_i. Equation 2.3 is used to produce the covariance matrix C[15, 25, 26].

$$C = \frac{1}{\mathrm{M}} \sum_{\mathrm{n}=1}^{\mathrm{M}} \Phi_n \, \Phi_n^T = AA^T \qquad \text{(Eq 2.3)}$$

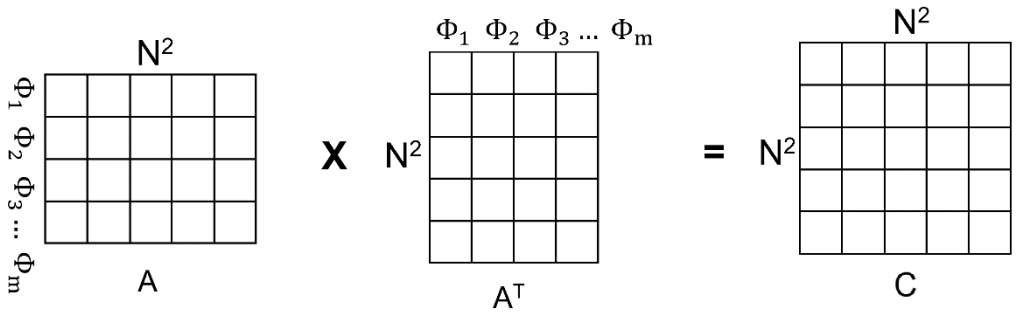

Fig. 2.7 A visual representation of how a covariance matrix (C) is calculated [15].

The covariance matrix, as seen in figure 2.7, is made up of all the images by combining the 1-dimensional vector (Φ) images. Note matrix C adopts the largest dimension from matrix A and A^T, denoted as N^2 for each matrix, thus making the covariance matrix very large. For example, if the images in the dataset are small, with pixel dimensions of 100x100, each image represents a single point on the covariance matrix. Hence each dimension is square (N^2) of the dimensions of the image, I_j. This makes the computation of this matrix very demanding and not practical. To solve this problem, the eigenvalues (λ) and eigenvectors (V_i), also known as eigenfaces, of the covariance matrix (C) are calculated. Equation 2.4 shows how the eigenvalues and eigenvectors are related to the covariance matrix C [15].

$$C \, . \, V_i = \lambda \, . \, V_i \qquad \text{(Eq 2.4)}$$

Calculating the eigenvectors of the covariance matrix gives more than is required. Instead, a relationship between the eigenvector (V_i) and matrix A is desired as this provides a more manageable number of eigenvectors. This is achieved by calculating the eigenvectors of the matrix A^TA, which has dimensions N x N [15, 25, 26].

Equation 2.4 can still be used to calculate the eigenvectors for A^TA, by replacing the covariance matrix C with it (equation 2.5). The eigenvector V_i is calculated using equation 2.6, by using the eigenvalue, λ [15, 25].

$$A^T . A\, V_i = \lambda\, . V_i \qquad \text{(Eq 2.5)}$$

$$V_i = \mathrm{A}\lambda \qquad \text{(Eq 2.6)}$$

The matrix A^TA and the covariance matrix C have the same eigenvalues but are the largest ones from the covariance matrix. The eigenvalues are then used to determine which eigenvectors are the most useful for facial recognition [15, 26]. The selected eigenvectors are known as eigenfaces. The image is used for facial recognition and converted into its eigenfaces. With the eigenfaces obtained, the images are placed into the face space (see figure 2.8 for an example of this). The location in the face space determines whether or not identification occurs successfully. Figure 2.8 [26] shows four possible outcomes (numbered 1 to 4) when an image is placed in the face space. The figure shows two eigenfaces (u_1 and u_2) and three known faces (Ω_1, Ω_2 and Ω_3). Outcome 1 is near the face space, and near a known face, hence the individual would be recognised. Outcome 2 is near the face space but not near a known face. This indicates the image is of an individual but is unknown. Outcome 3 is distant from the face space, and near a known face and outcome 4 is a face from both the face space and from known faces. Outcomes 3 and 4 would indicate that the image is not of a face [26].

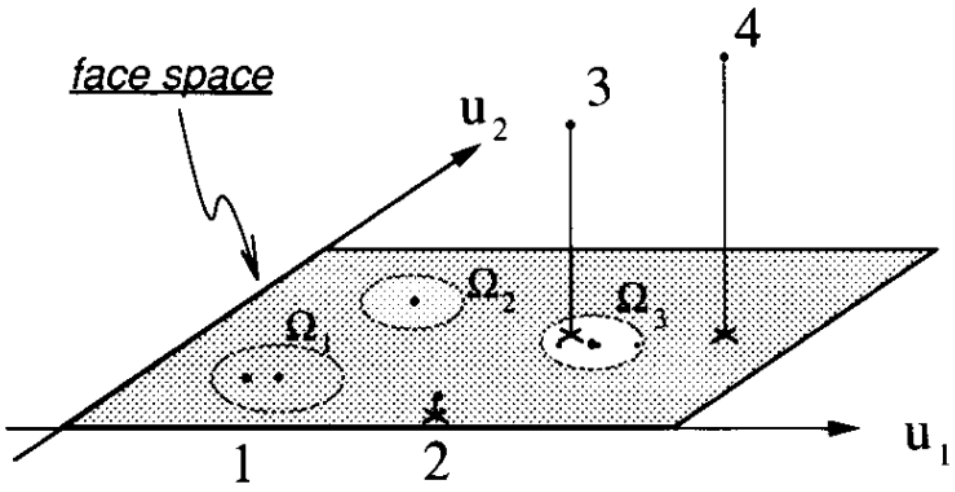

Fig. 2.8 An example of a face space [26].

2.4 Existing Technologies – Patents

2.4.1 Rearward Viewing Camera for Vehicles

Lynam [14] developed a vision display system for a vehicle that uses a video mirror display screen to show images of what is behind the vehicle. These images are captured by a rearward viewing camera of the vehicle while the vehicle is reversing. The driver can see a surround-view or bird's eye view representation of the vehicle, on a separate console, to avoid an accident while carrying out a reversing manoeuvre.

2.4.2 Multi-camera Image Stitching Calibration System

Lu [15] developed a vision system for a vehicle which included several cameras that have their field of view overlap one another. The system contains a display device which provides the driver with a display of images captured by the cameras of the vision system. The system includes the ability to allow user inputs to 'combine' images captured by the multiple cameras. This allows the system to align parts of the required target, found from each angle provided by each camera. The overlapping regions of the captured images of adjacent cameras are used to calibrate the cameras.

2.4.3 Detection of Driver Behaviours using in-vehicle Systems

DeRuyck [27] developed a system which included a motion sensor system configured for deployment in a vehicle which would develop and provide to the user a real-time digital mapping of driver movement during operation of the vehicle. DeRuyck's system includes an audio sensor which is triggered by specific driver movements, thus allowing for the provision of real-time feedback of their driving performance. The audio sensor is connected to a computer and motion sensor to provide feedback to the user. The feedback information includes providing data related to driver distraction occurrences by using driver movement mapping and the signal from the audio sensor. When distraction is detected, the computer would then provide an alert message to the provided user interface located in the vehicle.

Figure 2.9 shows a chart flow of how the system functions. Note how it involves the use of multiple sources of information to provide a more accurate analysis. Its inputs are driver context, driver history, driver behaviour, vehicle information, captured driver motion data and captured audio data. All of this is analysed to identify the driver's current behaviour.

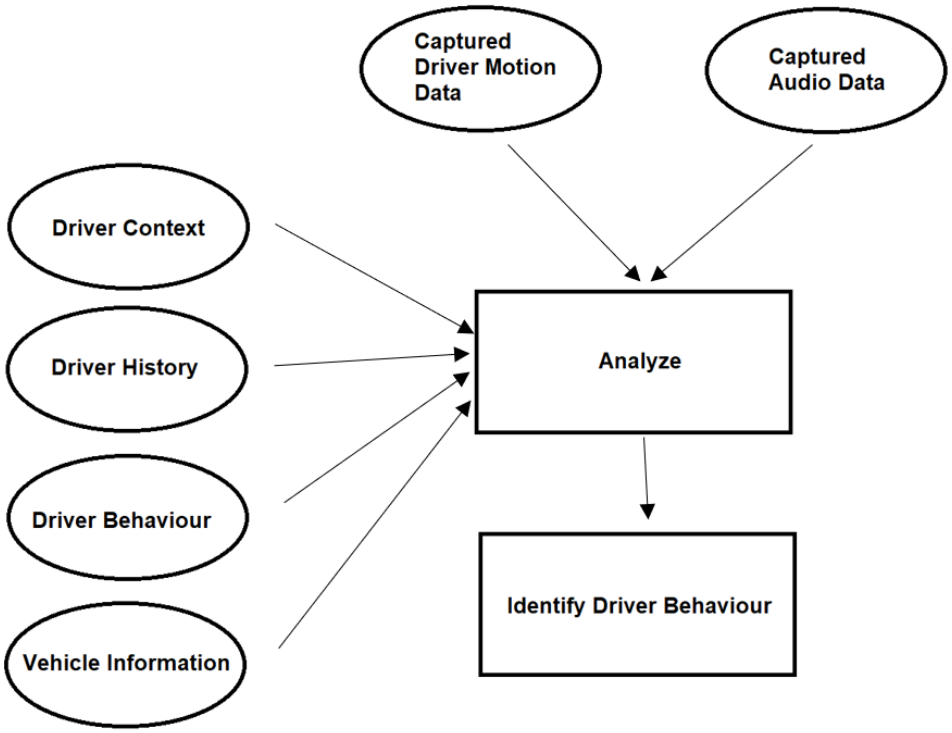

Fig. 2.9 Overview of Deruyck's system [27].

2.4.4 Multimedia Mirror System for Vehicles

Kramer *et al.,* [28] have developed a 'multimedia mirror system' that includes a display and a digital video playback device (e.g. MP4 player) which is attached to the rear-view mirror for ease of use and access, while maintaining their forward field of view via the windshield. While a multimedia system is irrelevant to the proposed system in this thesis, the way the system is implemented, and additional functionality are relevant and provide useful insight and ideas on how to implement best a system that is non-intrusive to driver behaviour.

2.4.5 Multiple Camera Vision Display System for Vehicles

Lynam [29] developed a vision display system for a vehicle which includes the use of multiple cameras which provide a field of view on all sides of the vehicle. The cameras feed a vision display system, that includes a display screen attached to the interior rear-view mirror. A second display screen is placed in a location in the vehicle away from the interior rear-view mirror of the vehicle, to allow the driver to view it while at the same times does not inhibit the driver's ability or performance when driving the car. The first display attached to the rear-view mirror has the primary use of displaying images captured by the rear viewing camera. These images include when the driver is carrying out a reversing manoeuvre of the vehicle. During reversing, the second display screen can provide a surround view, panoramic view or a bird's-eye view image, that is formed by combining the image data from the rest of the cameras attached to each side of the car.

The use of multiple cameras has enabled this system to provide real-time and accurate data with regards to the surroundings of the vehicle, all at the same time preventing distraction to the user while driving. This is primarily achieved by sensible positioning of the displays.

2.4.6 Drowsiness Detection System using Photoelectric sensors

Berezhnyy *et al.,* [12] developed a system that would detect drowsiness of a driver of a vehicle. Their specifications included a processing system that would accurately detect and monitor without needing high processing power, and can function just as well during low-light conditions at night time. The system can be installed in existing vehicles, whilst being cheap and affordable.

The system includes a processor, data interface and memory which is required to store data from sensors to be processed by the processor. The sensors involved are photoelectric sensors that compare changes in light level to determine movement, with respect to time. Using photoelectric sensor information to determine whether movement has occurred requires less processing power compared to video-based methods. This is due to video-based methods requiring feature detection and pixel comparison. Hence, photoelectric sensors can use less powerful processors and less memory, keeping costs low. Hence Berezhnyy *et al*.,

[12] used photoelectric sensors to determine movement, rather than camera-based methods. Note that one of their specifications was to produce a system that can be added to existing cars, and not just during assembly of a car's electronic systems. Hence the processing unit is independent of car systems and can be added without interfering existing car systems.

The system consists of a processing unit, interface to provide the user with visual information and photoelectric sensors to provide the data. Hence the system requires the photoelectric sensors to transmit light intensity data to the processor (transmitter), which in turn needs to be able to receive the incoming data as well as transmit the information to the interface (transceiver).

The data received by the interface has been described as relevant data that informs the driver about the vehicle's movement. The data received can be very basic, relating to movement, e.g. direction, or more detailed relating to movement data during a specified time interval. The data can also be processed to provide raw movement data to indicate driver drowsiness [12]. Note the different types of data the system can provide shows flexibility for the overall system. Looking more closely at the data the system provides to determine drowsiness, the system claims to be able to indicate not just the existence of drowsiness, but also the level [12]. The user is notified when a predetermined threshold is passed. The system proposed in this thesis uses a similar method with the threshold level being determined on an individual user basis.

Berezhnyy *et al.,* [12] used multiple photoelectric sensors rather than just one, as they found that it provided more accurate data regarding the vehicle's movement (figure 2.10). Having more than one photoelectric sensor allows the system to compare the data from each sensor to one another, allowing it to determine the direction of movement with respect to the light source to which the sensors are exposed [12].

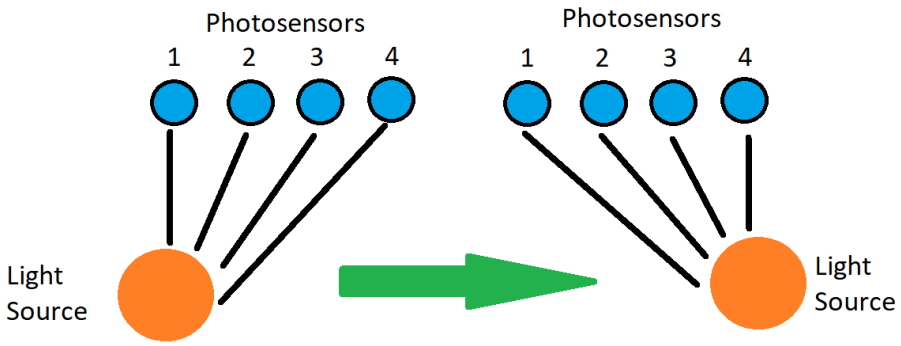

Fig. 2.10 Example of how multiple photoelectric sensors can provide movement data [12].

Looking at figure 2.10, the movement can be determined by comparing the light intensity between the photoelectric sensor receives from the light source as it moves across the linear sensors. The closer the photoelectric sensor is to the light source, the larger the value is and the further away the photoelectric sensor is, the smaller the value. Consider a situation where only photoelectric sensor 1 is available. Movement can still be detected since a change in light intensity can still be calculated. However, the presence of other photoelectric sensors allows for the direction to be detected as well. Hence, the light intensity change of photoelectric sensors 2, 3 and 4 allows for determining if the light source is on the left or right of the linear photoelectric sensors [12]. This also allows the detection of slight movements in either direction. However, this depends on how sensitive the photoelectric sensors are, which would be reflected in the overall cost [12].

The authors determined the following steps to measure drowsiness using photoelectric sensors. These steps included the following [12]:

1. Using multiple photoelectric sensors to transmit the light intensity they receive to the processor.
2. The light intensity data is used to determine the car driver's movements based on changes to the light intensity with respect to time. This follows the idea described and shown in figure 2.10.
3. The data are analysed to determine the changes in movements and their relation to patterns indicating signs of drowsiness.

The data are analysed by the processor first to determine the position of all the photoelectric sensors providing light intensity data, with respect to the light source

[12]. Second, it must maintain the changing position as the light source moves further or closer to the photoelectric sensors. This allows the system to provide real-time and accurate information [12]. With regards to determining drowsiness, this is determined by the change in position of the photoelectric sensors with respect to light source [12]. The comparison itself requires knowledge of previous positions of the photoelectric sensors with respect to the light source. Hence these previous positions would need to be stored in memory to be accessed later for this comparison to occur [12].

Note that the amount of memory required for storing this specific set of data is dependent on how far back the user wants to store this data for [12]. However, in general, the amount of memory required is much more when using video imaging sensors, compared with photoelectric sensors [12]. This method has an advantage over the proposed method in this thesis, in relation to the amount of memory required [12] because the system proposed in this thesis uses computer vision techniques which need more memory space than photoelectric sensors. However, an advantage of the proposed system has is it looks at signs of fatigue/ drowsiness on the face of the user, [12] looks at how the user is driving, and bad driving does not necessarily imply fatigue.

Berezhnyy *et al.,* [12] improved on the accuracy of the data provided by the photoelectric sensors by using ones which picked up on patterned light, for example detecting the colour information of the light hitting the sensor. The light flicker frequency of the patterned light would have to be so high that the human eye cannot distinguish the flickering. This is vital, so to not distract the driver. With the light source having a specific pattern that the system can identify, it can also disregard light from other sources (equivalent to noise) [12].

Another requirement they considered is the light source providing infrared light. The advantage of this is that human eyes do not detect infrared light, and so will not distract the driver [12]. Infrared light can also be detected in all light levels, including deficient light levels, such as during night-time [12]. This is even more important for this system, as drowsiness is most likely to occur/increase during night-time [12].

In terms of the overall system speed, the information used and analysed is the comparison of light intensity being read by each photoelectric sensor [12]. Hence

the data can be used directly, and so leads to a faster and more reliable outcome. This is due to the system relying on accurate information, which is dependent on the accuracy of the photoelectric sensors, and the ability to quickly analyse the incoming data means more accurate real-time data [12].

2.4.7 Driver Distraction and Drowsiness Warning and Sleepiness Reduction for Accident Avoidance

Mimar [13] developed a system which monitors the driver to avoid accidents which could be caused by drowsiness or distraction. The system measures the driver's distraction and fatigue levels by looking at the driver's face and continuously tracking it with the speed and maximum time allowed. When drowsiness or distracted behaviour is detected, a noisy alert and a dim blue light are triggered, which assist in waking up the driver.

The system achieves this by using multiple camera sensors, where image signal processing (ISP) is carried out on each sensor [13]. Facial recognition is carried out to monitor the driver to detect driver distractions and drowsiness. The distraction is measured by looking at the direction of the driver's face. Measurements regarding this are done with respect to speed and cornering of the vehicle. The driver's drowsiness level is measured by looking at head angle and also the position of the eyelids. Once again, when either drowsiness or distraction of the driver occurs, the system sets off an alarm to notify the driver. The system goes a step further by providing the user with audio and video recordings of the journey to allow the driver to review their own levels of drowsiness and how distracted they get. This allows them to try to discover what the cause is and how to improve their safety while driving.

2.4.8 Doze Detection Method

Nakamura *et al.,* [30] developed a system that can accurately detect a blink burst which can improve the speed and accuracy of doze detection. It can also measure when the eye is wide open and when the eyes are fully shut. The system looks at comparing the average blink interval of an adult and comparing it the average time the eye is closed. By looking at these parameters, the system detects a dozed

state when the current time the eye is closed is more than the standard average time.

This implies the user is aware of the drowsiness levels and is trying to blink quickly to stay awake but cannot help but keep their eyes shut for long periods. When this is the case, the system would then trigger an alarm.

2.4.9 Face Recognition System

Son *et al.,* [31] developed a system to control a mobile terminal by detecting a face or an eye in an input image which the user provides. They developed a method that included performing facial recognition on an image that the system takes of the user. The face recognition used also provides data such as whether the user exists in the system memory or if it is a new user, the direction of the user's face, how far from the mobile terminal the user is, and the exact position of the user's face.

The system can also look specifically for the user’s eyes to control the mobile terminal. This was developed as a backup in case the face cannot be detected.

2.4.10 Drowsiness Detection System to Detect Drowsiness

Nilsson *et al.* [32] developed a drowsiness detection system to detect drowsiness levels of the user. The system involves the use of a digital camera and a control unit. The digital camera is used to detect the eyes of the user by taking a series of images. These images are used by a processor to detect eyelid movements. The eyelid movements are converted to an electrical signal which is sent to the controller unit. This unit converts the raw data of the electrical signal into a modelled signal. The comparison looks to see if the modelled signal represents a blink. To accurately determine if a blink has occurred, the controller uses multiple modelled signals with respect to time. If the level of alertness falls below a threshold, an alarm is triggered.

2.4.11 Driver drowsiness detection

Yang [33] developed a driver drowsiness detection system for Ford Global Technologies Data. While the systems previously discussed looked at blink rate or vehicle positioning to determine drowsiness levels, this system uses a different approach by looking at the yaw-rate of the vehicle, with respect to time. Figure 2.10 shows a flow chart describing the process.

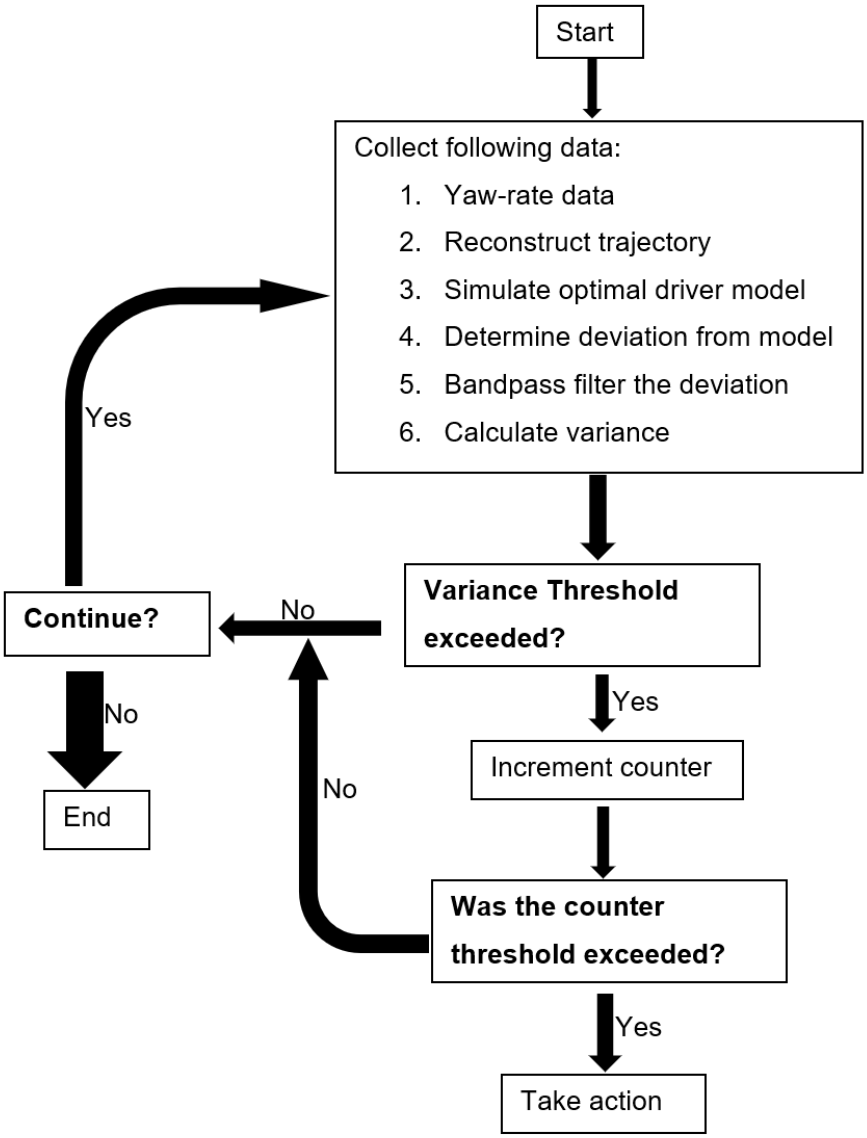

Fig. 2.11 Flow chart of how the yaw rate is used to determine driver drowsiness levels [33].

The system starts by collecting data from sensors (yaw rate sensor), simulation results model data. From this, it calculates the variance which is then compared to a threshold value which determines whether the counter is incremented or not. The variance is calculated when looking at the actual trajectory and ideal trajectory using the yaw rate sensor data. Suppose the threshold exceeds, the counter would increase. If the counter surpasses its own threshold, then the action is taken. If the variance threshold or counter threshold are not exceeded, then the system asks if the algorithm should continue or end. If to continue, then the system returns to the data collection phase.

2.5 Summary

The techniques described in this chapter used various methods to meet their requirements. Some relied on detection of facial features so they can use them to measure fatigue levels, whilst others used driving technique as a measurement of fatigue

The advantages and disadvantages of each approach were discussed, along with a brief technical discussion of the techniques. This provided an overview of state of the art in fatigue detection, which informed the author regarding suitable and non-suitable approaches, as well as showing the opportunities for original contribution.

I concluded that the best and most original approach would be to use a Neural Network to process the image data. In chapter three, I show this is more efficient and reliable compared to more traditional methods such as feature detection described in this chapter.

2.6 References

[12] I. Berezhnyy, G. N. Garcia Molina, and M. A. Weffers-Albu, "Detection system using photo-sensors " United States of America Patent 9,573,598, 2017.

[13] T. Mimar, "Driver distraction and drowsiness warning and sleepiness reduction for accident avoidance " United States of America Patent 9,460,601, 2016.

[14] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, 2001, vol. 1: IEEE, pp. I-I.

[15] M. Turk and A. Pentland, "Eigenfaces for recognition," *Journal of cognitive neuroscience,* vol. 3, no. 1, pp. 71-86, 1991.

[16] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau, "Real-time eye blink detection with GPU-based SIFT tracking," in *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, 2007: IEEE, pp. 481-487.

[17] "Cascade Classifier." The OpenCV Reference Manual. http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html (accessed July 2017).

[18] S. Soo, "Object detection using Haar-cascade Classifier," *Institute of Computer Science, University of Tartu,* pp. 1-12, 2014.

[19] R. Padilla, C. F. F. Costa Filho, and M. G. F. Costa, "Evaluation of haar cascade classifiers designed for face detection," *World Academy of Science, Engineering and Technology,* vol. 64, pp. 362-365, 2012.

[20] R. E. Schapire, "The Boosting Approach to Machine Learning An Overview," *Nonlinear Estimation and Classification,* 2001.

[21] G. Bradski, "The OpenCV Library," ed. Dr. Dobb's Journal of Software Tools: OpenCV, 2000.

[22] R. E. Schapire, "Explaining Adaboost," in *Empirical inference*: Springer, 2013, pp. 37-52.

[23] M. Yang, J. Crenshaw, B. Augustine, R. Mareachen, and Y. Wu, "AdaBoost-based face detection for embedded systems," *Computer Vision and Image Understanding,* vol. 114, no. 11, pp. 1116-1125, 2010.

[24] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman, "Eigenfaces vs. fisherfaces: Recognition using class specific linear projection," *IEEE Transactions on pattern analysis and machine intelligence,* vol. 19, no. 7, pp. 711-720, 1997.

[25] M. Çarıkçı and F. Özen, "A face recognition system based on eigenfaces method," *Procedia Technology,* vol. 1, pp. 118-123, 2012.

[26] M. Turk and A. Pentland, "Face recognition using eigenfaces," 1991, pp. 586-587.

[27] J. DeRuyck, "Detection of driver behaviors using in-vehicle systems and methods " United States of America Patent 9,714,037, 2017.

[28] M. E. Kramer, N. R. Lynam, D. P. O'Connell, and V. R. Nise, "Vision system for vehicle," United States of America Patent 9,632,590, 2017.

[29] N. R. Lynam, "Vision display system for vehicle " United States of America Patent 9,598,014, 2017.

[30] K. Nakamura and H. Takano, "Doze detection method and apparatus thereof " United States of America Patent 9,286,515, 2016.

[31] B.-J. Son, H.-I. Kim, and T.-H. Hong, "Apparatus and method of controlling mobile terminal based on analysis of user's face," United States of America Patent 9,239,617, 2016.

[32] B. Nilsson and E. Rosen, "Device and method for detecting drowsiness using eyelid movement " United States of America Patent 9,220,454 2015.

[33] Yang, Hsin-hsiang, and K. O. Prakah-Asante, "Driver drowsiness detection " Patent 9,205,844, 2015.

Chapter 3: Neural Networks

3.1 Introduction

With conventional techniques discussed in chapter two, chapter three focuses on Neural Network techniques. Neural Networks are designed to determine the relationship between the output and its respective input. This process is known as the training phase. This chapter discusses different types of learning that can be done, which is dependent on the type of training dataset. For the proposed models in chapter five, the type of learning used in my research was supervised learning. Consider teaching a child to identify different shapes. The process would involve showing the child a shape, for example, a square and correctly informing them that what they are seeing or identifying is a square. This type of data is considered labelled and supervised since the child is being informed on what the correct label is (square).

The aim of this type of learning is for the model to use training to learn the relationship between the inputs (images) and the output (the corresponding label). There are, however, various factors which determine the effectiveness of the training, and subsequent validation of the dataset being used. These are discussed in this chapter, along with Convolutional Neural Networks (CNNs), which is the type of Neural Network the proposed algorithm and models were designed on.

3.2 Machine Learning Algorithms

Machine learning algorithms can, in most cases, be categorized into four categories, which are supervised, unsupervised, semi-supervised and reinforcement [34, 35]. Table 3.1 summarises each type of algorithm with the respective data type they use.

1. Supervised machine learning algorithms work by applying labelled data which are used to allow the algorithm to learn the relationship between the input samples and output labels. Labelled data is data that is assigned a tag or label, allowing the algorithm to identify the data. For example, an image of a cat will have a label that the algorithm has access to which informs it the image is of a cat. Once this relationship is learned, it is then used to predict future events [34, 36]. This is achieved by allowing the algorithm to analyse the known training dataset that allows it to predict output labels. The learning algorithm can also compare its output with the expected output since it uses labelled data. This accuracy can then be used to locate errors and change the algorithm to improve accuracy [34, 36].

2. Unsupervised machine learning algorithms use datasets which are neither labelled nor classified. Hence the samples in the data set are not tagged. For example, labelled data which consisted of an image might tag certain features in the image, making it labelled data. Unlabelled data would not have these tags. These data sets allow the algorithm to develop a function that defines the relationship from the data, between the inputs and outputs. The output is not correctly produced from this, but what is produced, is the algorithm discovering hidden patterns from the unclassified data sets [34, 36].

3. Semi-supervised machine learning algorithms use both labelled and unlabelled data for training, usually a ratio that has a high proportion of unclassified data. Using both types of data the accuracy of the algorithm usually supersedes the two algorithms described above This is typically used when studying how a system learns when exposed to both types of data (unlabelled and labelled) [34, 36, 37].

4. Reinforcement machine learning algorithms (see figure 3.1) typically contain two components, an agent and an environment. The agent represents the algorithm, and the environment is the data that the agent uses to update and improve the relationship the algorithm is based on. The environment component (figure 3.1) has two outputs. One output is the reward output, while the other is the state output. The state output is the current state or situation of the agent, while the reward is the feedback from the environment [38].

5.

Table. 3.1 Data used by each algorithm category.

Categories	**Data used**
Supervised machine learning	labelled
Unsupervised machine learning	unlabelled
Semi-supervised machine learning	labelled and unlabelled
Reinforcement machine learning	Continuously learns from data.

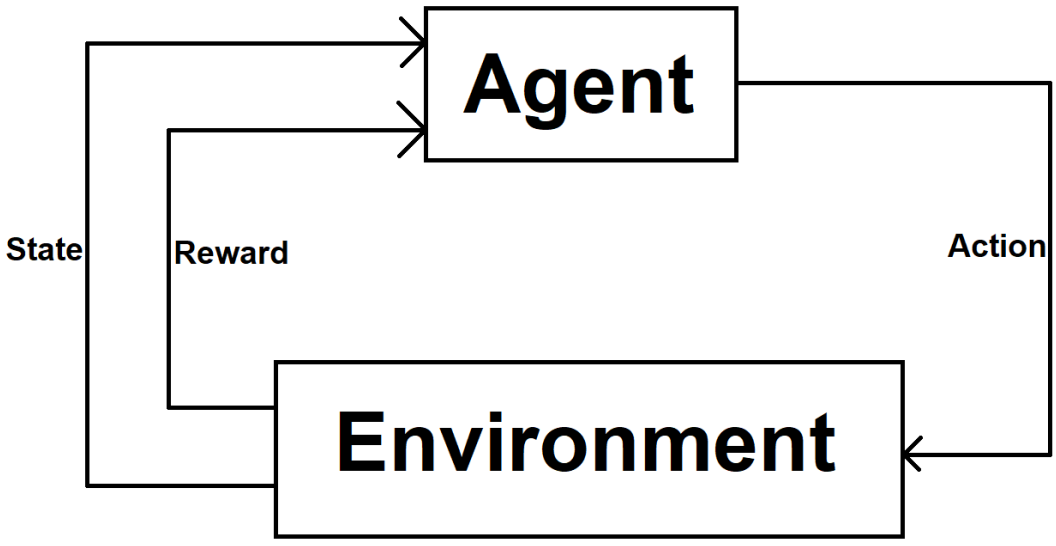

Fig. 3.1 Reinforcement learning algorithm example [38].

While the methodology of these algorithms differs, their purpose and outcome are similar, in that they allow systems to analyse vast quantities of data. For example, when detecting the blink rate or driver distraction.

With the algorithms described above, it is crucial to understand how Neural Networks work, and subsequently, how the relationship between the input and output is defined [35]. Neural networks work in a similar way to brain neurons [35]. Figure 3.2 shows an example of how a Neural Network might be connected. There are three main areas in Neural Networks, the input layer, the output layer and a hidden layer. The hidden layer is where the relationship between the input and output is defined. A Neural Network technique known as backpropagation, similar to reinforcement machine learning, uses errors in the output to adjust its algorithm by adjusting the hidden layers.

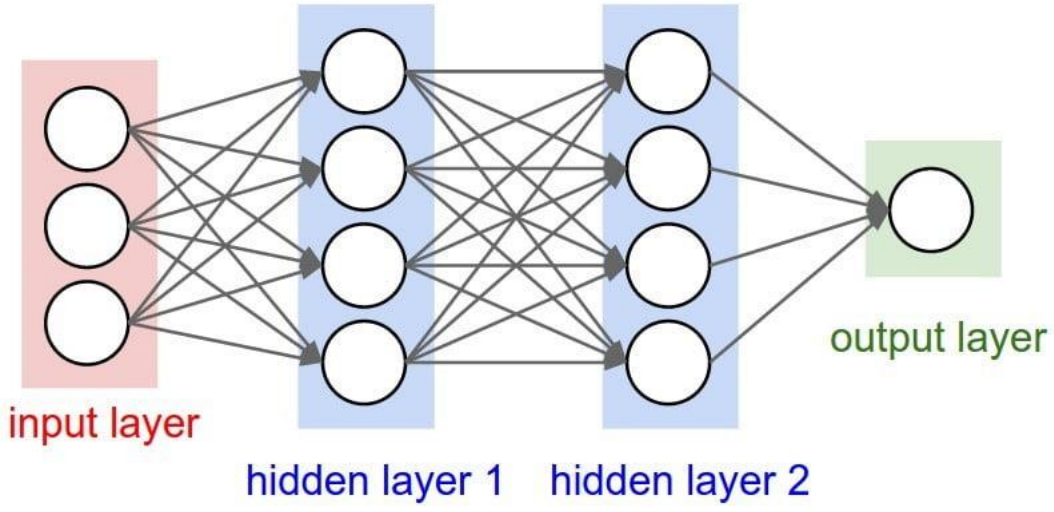

Fig. 3.2 Example of how the three layers are connected to each another [39].

The basic principle of how Neural Networks function is that the data set (input layer) is passed to the hidden layers, of which each will extract a different set of features with every layer [35]. For instance, with my algorithm, the first hidden layer looks at the blink rate and the subsequent ones after looking at head tilt and how long the eyes are shut. By the time all the layers have been applied to the input data set, the algorithm would then be able to create a complex fatigue detector. The same principle can be used to detect other outcomes, for example detecting facial

features. This process is known as the training section. Once complete, the output can be labelled, and any errors can be rectified using backpropagation. As the algorithm corrects these errors, it will be able to classify the tasks it is designed for without human intervention involving manually changing the algorithm [35].

3.3 Types of Neural Networks

This section discussed the different types of Neural Networks that exist and that were explored as I worked on designing the algorithm. Firstly, is feedforward Neural Networks (figure 3.3). This is a basic form of a Neural Network in which the information travels from the input layer to the output layer in one direction [35]. This version has no feedback connections whereby the output layer is fed back into the input layer [40].

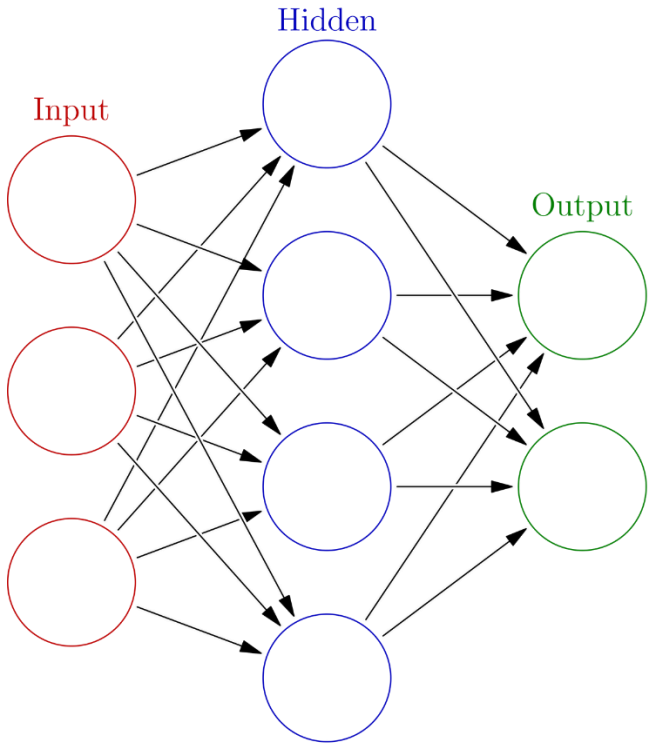

Fig. 3.3 Feedforward Neural Network [41].

This form of a Neural Network can have a hidden layer, but also the connections between the layers can be partially connected or fully connected (also known as Dense).

The more commonly used network, as mentioned above, is the recurrent Neural Network, where data flows in multiple directions [35]. Figure 3.4 shows how a recurrent Neural Network looks. The reason for their increased popularity, when compared to, feedforward Neural Network (see recurrent network label in figure

3.4), is they have increased capability since they can allow feedback into the input. Therefore, the network can learn from the output to improve the algorithm.

In figure 3.4, the recurrent Neural Network does not necessarily have to feed the output back into the input. All that is required is any layout (with the exception with the input layout) feeding back into a previous layout. In the case of figure 4, the feedback occurs between the two hidden layers [35].

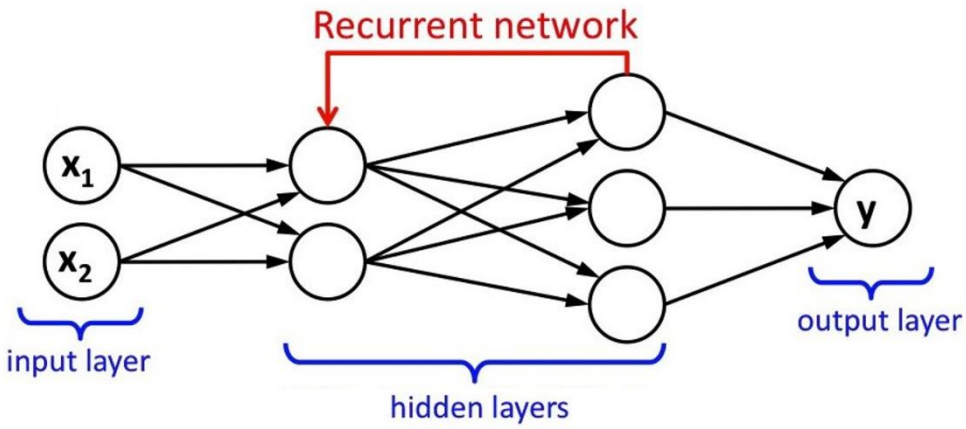

Fig. 3.4 Recurrent Neural Network [42].

Convolutional Neural Networks work with input images by assigning weights and biases to specific objects in the image. These are used to identify and differentiate between these objects [43]. For instance, in my case, the object in question is the eyes and the borders of the head. The algorithm shares very similar aspects to feature recognition with kernel weights.

Restricted Boltzmann machine networks (RBM) are shallow, two-layer Neural Networks (see figures 3.5 and 3.6). The first layer of the RBM is called the visible, or input, layer, and the second is the hidden layer [44]. Each circle in figure 3.5 is a node acting like a neuron. The nodes are connected to each other across the layers, but not two nodes of the same layer are connected (intralayer communication). It is this that makes it a restricted Neural Network [44].

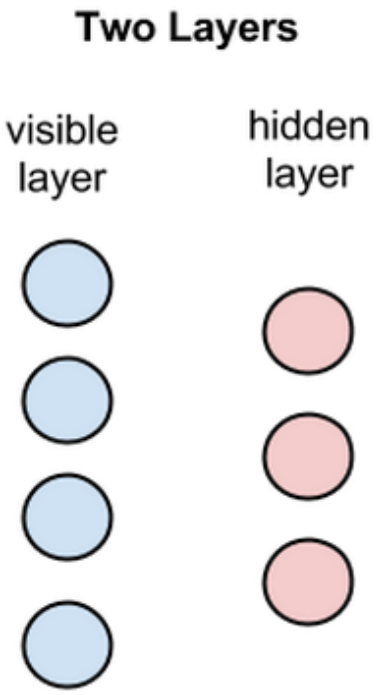

Fig. 3.5 Layers in a Restricted Boltzmann machine network [45].

Starting with the visible layer, each node in this layer takes a feature from an item in the data set [44]. If looking at an image, each node would have one pixel-value for each pixel in one image. So, if an image has 500 pixels, then the visible layer (input layer) would need 500 input nodes. The relationship and representation of the layers are what produces a function that describes this relationship. Another way this algorithm could work is if several inputs would combine at one hidden node. For example, each input x is multiplied by a weight, and the products are summed and added to a bias, b. the result here is passed through the activation function to produce the node's output.

Another alternative is to fully connect (dense) each node in the visible layer to all the nodes in the hidden layer. When this occurs, the Restricted Boltzmann machine network (figure 3.6) can be described as an asymmetrical bipartite graph [44]. Each input x (figure 3.6) is connected to each hidden layer node, and thus would be multiplied by each of the weights. Hence each input is subjected to 3 weights from the hidden layer. A summary of different types of a Neural Network can be viewed in appendix 8.1.

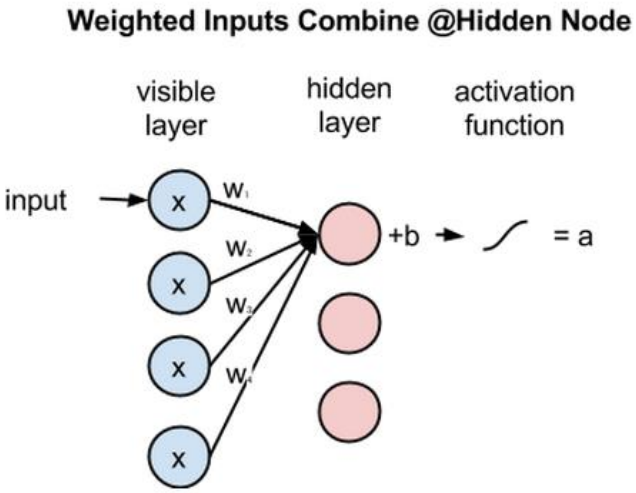

Fig. 3.6 Restricted Boltzmann machine network with convergence on one node [45].

3.4 Learning Aspect

Neural networks need data to develop an algorithm that defines a very accurate relationship between the input and output. There is a direct correlation with an increase in data provided for learning to more accurate outputs. The data that is provided is broken into three sections. Before this, what usually happens is that there is a random selection of data beforehand [46]. For example, say the data set contained 100,000 pieces of data. 50,000 samples of this data set are selected at random. From these selected samples, roughly 80% are used for the first stage of training, followed by 10% for validation and the final 10% for testing.

3.5 Convolutional Neural Networks

Convolutional Neural Networks take an input image and assign weights to various objects in the image. This is done to differentiate between the objects in the image [43].

3.5.1 Input Image

Figure 3.7 represents an RGB image which has been separated into three planes representing red, green and blue. Convolutional Neural Networks break up the image to process it faster, and without losing features which are critical for getting a correct prediction [43]. This feature is essential for my algorithm as it allows processing large data sets.

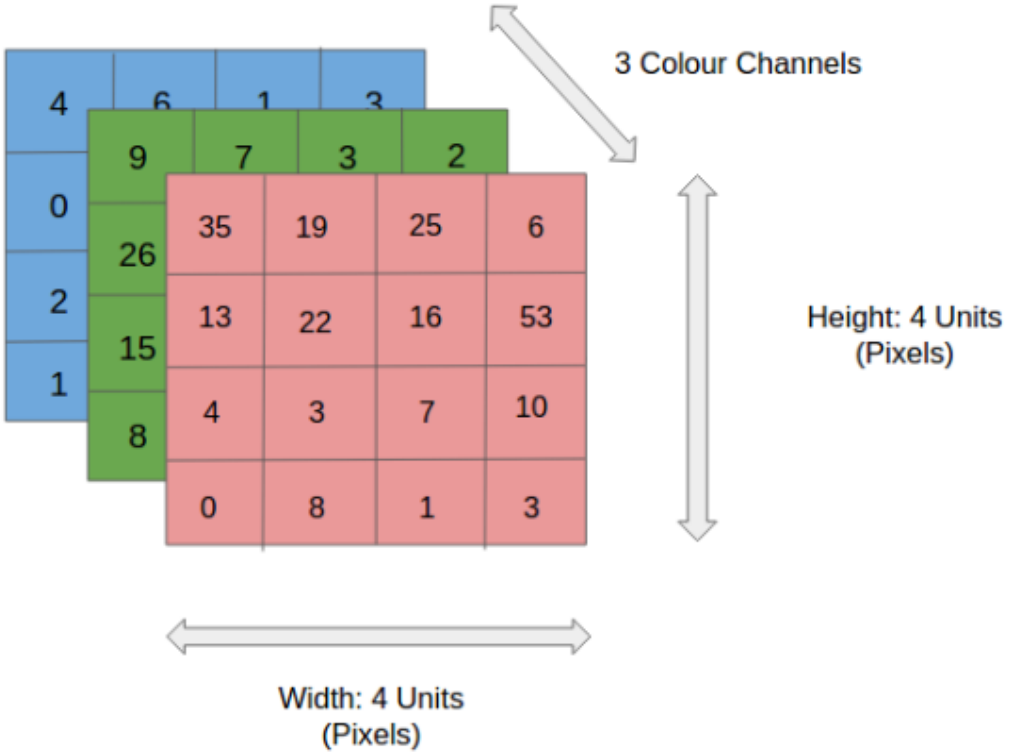

Fig. 3.7 Example of a 4x4x3 RGB image [47].

3.5.2 Convolution Layer

To describe how convolution works, I will be using the 5x5 image shown in figure 3.8. The yellow box is the kernel, which is used to carry out the convolution operation in the first part of a Convolutional Layer. While the kernel can be any dimension, in figure 3.8, the dimension of the kernel used is 3x3, also known as the convolved feature.

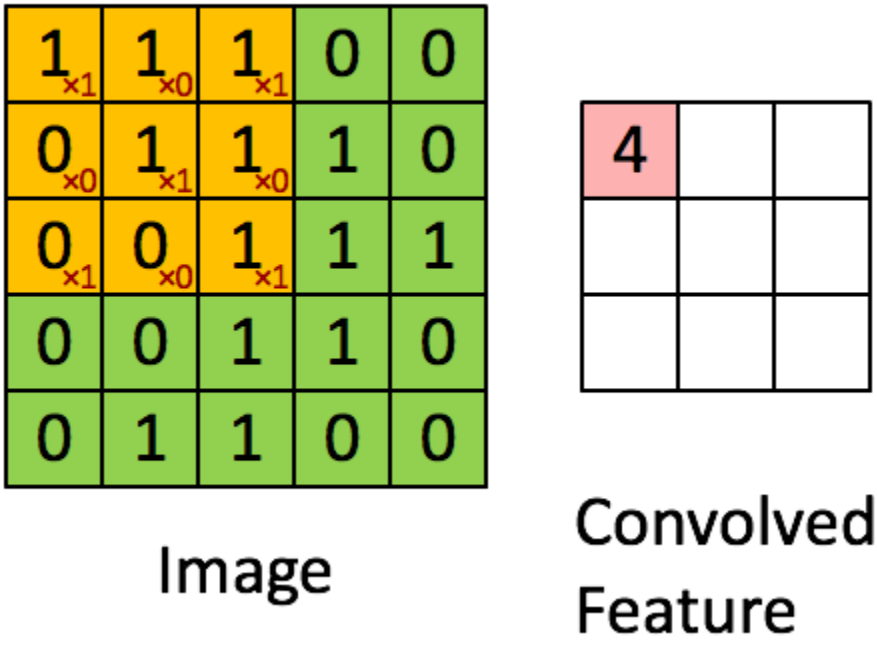

Fig. 3.8 How to convolute an image with a kernel to get a convolved feature [47].

The green section resembles the 5x5x1 input image, and the yellow section is the kernel. The filter moves over the image, across one column at a time. A matrix multiplication operation is calculated between fennel filter and the area it covers on the image [43].

The filter moves across the image, starting in the top left corner, and makes its way to the right by a certain number of squares/pixels. Once it reaches the end of the image, it moves down a level and continues, from the left side of the image, again. This process is repeated until the entire image is completed [43].

3.5.3 Purpose of the Convolution method

Convolution methodology extracts the high-level features from an input image, for example, an edge that would separate two features in the image. Convolutional Neural Networks can have multiple Convolutional Layers. In these circumstances, the first Convolutional layer, in the Neural Network, is used to locate Low-Level features, like edges, different colours or gradient orientation[43]. The next convolutional layers will look at higher-level features, providing a more detailed understanding of what is contained in the image. Therefore, the more layers used, the better understanding of the image is provided [43].

3.5.4 Results from Convolution

There usually are two outcomes of convolution.

1. Valid Padding - dimensionality is decreased compared with the input.

2. SAME Padding - The dimensionality is increased or remains the same when compared to the input.

The second outcome is achieved by padding the image with zeros. This is achieved by adding zeros are added to the edge of the image matrix to expand it. Figure 3.9 shows what padding looks like. Note the grey squares are where the zeros are placed for the expansion.

The padding can be done before the kernel or filter is applied. So, for example, the size of the original 5x5x1 image can be increased into a 6x6x1 image using padding, then apply the filter over it. The outcome is a convolved matrix of size 5x5x1, which is the same as the original image. Hence this is known as SAME padding [43].

Without padding, the output will be the same dimension as the filter, so if a 3x3x1 filter is used, then the output image will be of size 3x3x1. This is known as Valid Padding, where the output image will always be smaller than the input image [35].

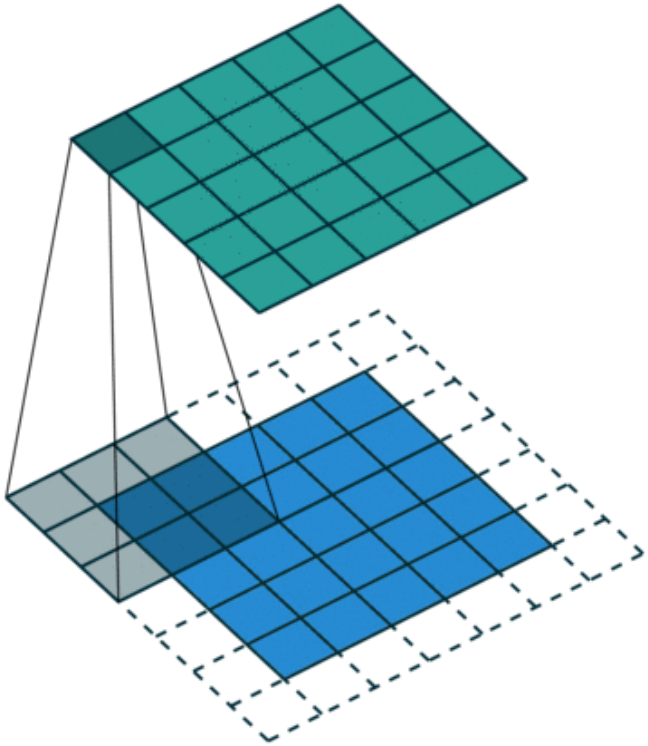

Fig. 3.9 A 5x5x1 image is padded to create a larger, 6x6x1 image [47].

3.5.5 Max Pooling and filtering

During image processing, the location of the desired features in the image can confuse the learning of convolutional networks [48, 49]. Take an example of a child learning what a cat is. When showing the child images of cats, the position and posture of the cat in the images will not confuse the brain of the child. If we present images of cats in different positions and postures, a machine learning system can be easily confused by the different postures and positioning of the cat. This confusion can be solved by downsampling the feature map. The result of this will mean the machine learning system will be more robust against the localisation of the desired features.

Downsampling can be achieved through Pooling. In Convolutional Neural networks, each convolutional layer is immediately followed by a pooling layer. The convolutional layer is connected to the pooling layer, typically with a non-linear activation function (discussed later in chapter three). The pooling layer works

similarly to convolution. The input of the pooling layer is the output feature map produced by the convolutional layer that is connected to it [48, 49]. The pooling layer applies a filter (which is smaller than the convoluted feature map). Typically, the applied filter is of size 2x2 pixels, which was used in the proposed models in chapter five. The size of the filter and the stride (the number of pixel jumps the filter makes across the image horizontally and vertically) determines how much the convoluted feature map is downsampled by. A 2x2 pixel filter and stride of (2, 2) will downsample the convoluted feature map by half in both dimensions of the image. The result of this will mean the Neural Network model will be less sensitive to the positioning of features in the input image [48, 49].

There are two approaches to pooling, which are listed and described below:

- Average Pooling is based on calculating the average value of each filter as it moves along the image. This average value in the filter is used as the new pixel value in the output image after Average Pooling [48, 49].
- Max Pooling is based on calculating the maximum value of each filter. This maximum value in the filter is used as the new pixel value in the output image after Max Pooling [48, 49].

Figure 3.10 shows an example of Max and Average Pooling, using a filter size of 2x2 pixels and a stride of 2x2 pixels [48, 49]. The figure clearly shows the size of the output Pooled image is reduced by half, due to the stride of 2x2 pixels, meaning no pixel in the input Convoluted image was Pooled more than once.

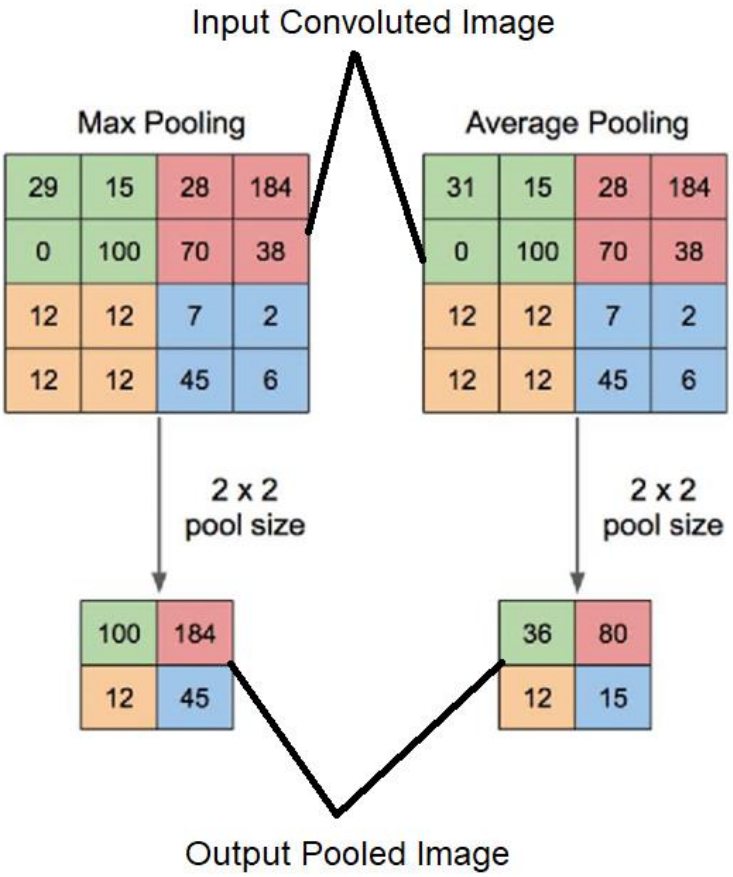

Fig. 3.10 Examples of Max and Average Pooling with filter size 2x2 and stride 2x2 [48].

3.6 The architecture of Deep Learning and Neural Networks

This section aims to provide an explanation of various terminology used when developing a Neural Network. While different libraries can be used to design a Neural Network, I chose to use the Keras library to develop the Neural Network proposed in this thesis [50].

Keras [50] is an open-source library that allows the user to produce Neural Network models. The library itself was written in python and can run with TensorFlow [51]. When explaining the various terminology, I will be using the Keras library [50] to help explain and share what it can offer the user.

Neural network models use many variables, such as the number of epochs, neurons in a layer, the total number of layers, the size of the kernel and learning rate, among others [46]. Small changes to these variables have an impact on the accuracy at the output. However, the gains or reductions in the accuracy have the side effect of increasing or decreasing the execution time of the learning process.

3.6.1 Activation functions

The output of any neuron in a Neural Network is the weighted sum of all the input signals to that signal. Without an activation function, the value of the output of each neuron will exceed one. The output of neurons in the hidden layer and the input layer contributes to the output of the neurons in the output layer. The output of each neuron in the output layer corresponds to the probability one label. The number of neurons in the output layer is equal to the number of labels in the dataset. With no activation functions in the hidden layers, the output of the neurons in the output layer may exceed one. This will increase the likelihood of Neural Network predicting the label of each input image incorrectly. Thus, activation functions are used in Neural Networks to scale down the output of neurons in the hidden layers [46, 52].

Figure 3.11 shows an example of an activation, f, applied to the sum of the weighted inputs and bias, b, output of a neuron. The input neurons are x_1 to x_n, with their respective weights of w_1 to w_n [53]. Table 3.3 lists the activation functions that can be accessed with the Keras library [50], which the Neural Network,

presented here, uses. The available functions are Linear, Sigmoid, Tanh, SoftMax and ReLU [46].

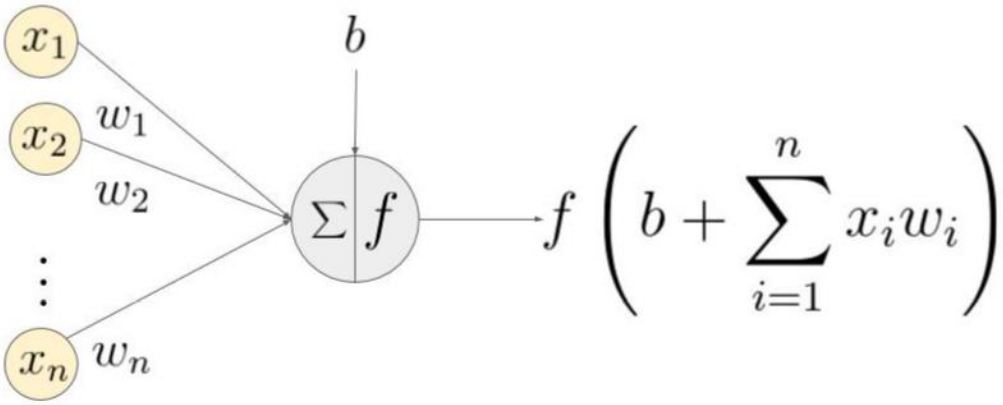

Fig. 3.11 Activation function applied to the output of a neuron [53].

Table. 3.2 List of activation functions available for use.

Linear	Basic activation function. The signal does not change. The input and output relationship is linear [46]. $$y = mx + c$$
Sigmoid	Reduces extreme or atypical values within valid data, without eliminating them. It converts independent variables of almost infinite range into simple probabilities between 0 and 1. Most of its output will be very close to the extremes of 0 or 1 [46]. $$\sigma(x) = \frac{1}{1 + e^{-x}}$$
Tanh	Represents the relationship between the hyperbolic sine and the hyperbolic cosine. The equation for Tanh is: $$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$ The normalized range of tanh is between -1 and 1. The advantage of tanh is that negative numbers can be dealt with more efficiently [46].
Softmax	The softmax activation function is typically found in the output layer of a Neural Network. It returns the probability distribution over mutually exclusive output labels. Hence the label with the highest probability is selected to assign the label [46]. $$S(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$
ReLU	The activation function rectified linear unit (ReLU) checks a predefined threshold. Once this threshold has been met, it activates a single node. The output of ReLU is 0 if the input value is below 0. When the input rises above 0, the output follows a linear relationship with the input variable of the form [46] $$f(x) = x.$$

3.6.2 Bias

A bias value to a neuron allows the activation function to be shifted left or right, which may be critical for successful learning. Figure 3.12 shows a function that calculates the output, using various values of weights. The output of a neuron is calculated by multiplying the input (x) by the weight (w_0) and passing the result through an activation function [54, 55].

Changing the weight, w0, fundamentally changes how steep the activation function is. To produce an output of zero with a non-zero input, changing the steepness of the activation function will not work. However, this can be achieved by shifting the activation function left or right. This is the role of the bias in a Neural Network. The bias is a constant value that acts like the y-intercept in a straight-line equation. The y-intercept allows the straight-line function to shift across the x-axis to make y (dependent variable) equal zero for a non-zero x (independent variable). Figure 3.12 shows the bias affecting the activation function in the same manner as the y-intercept when compared to the non-bias function in figure 3.12 [54, 55].

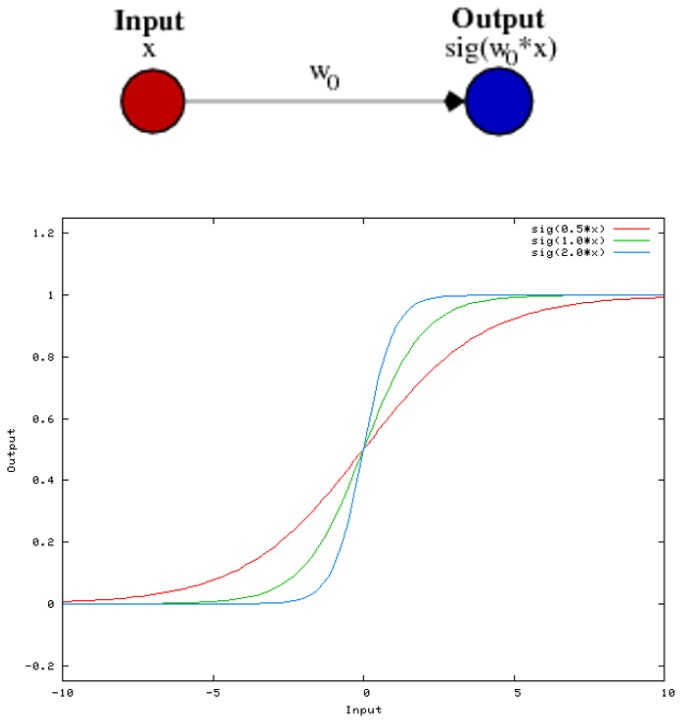

Fig. 3.12 Example of a sigmoid activation function without bias [54].

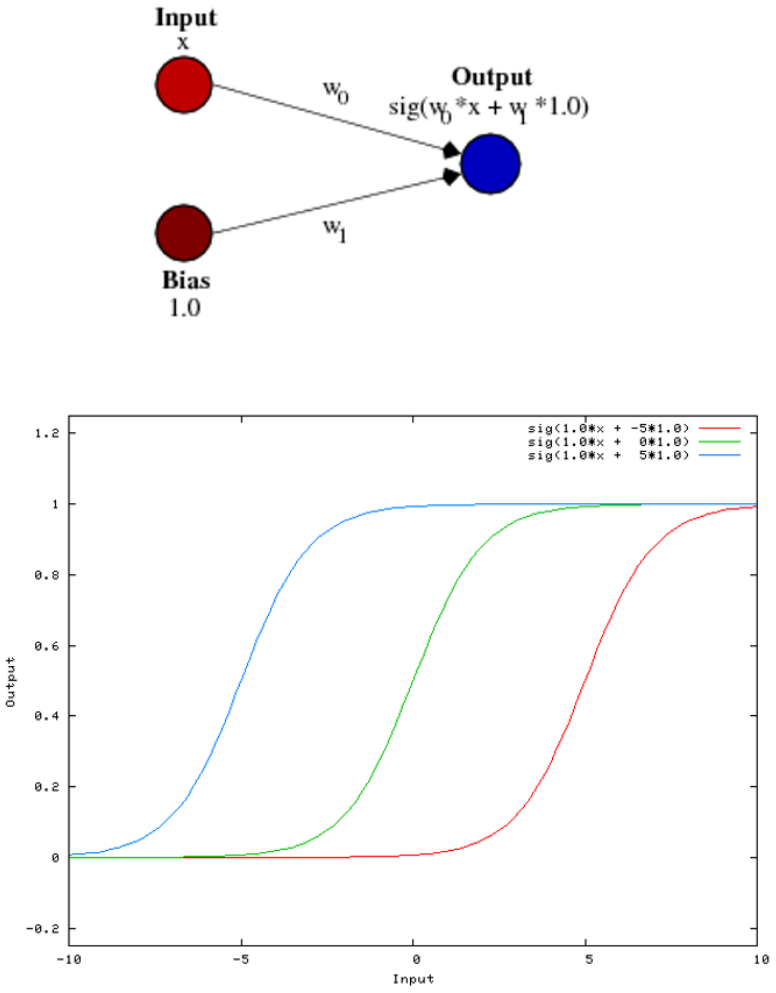

Fig. 3.13 Example of a shifted sigmoid activation function with bias [54].

3.6.3 Parameters and hyperparameters

Parameters are configurable variables that are internal to their function. In this case, a Neural Network model, where their value can sometimes be estimated from the data [56].

A hyperparameter is a configuration variable that are external to the model itself and whose value, in general, cannot be estimated from the data and are specified by the programmer to adjust the learning algorithms. The key to an efficient Neural Network is finding the most optimum hyperparameters, which must be specified before starting the training process so that the models train better and more quickly. These parameters will be the focus when investigating the optimum values during the training of the Neural Network [56].

3.6.4 Epoch and Batch Size

An epoch can often be misinterpreted as an iteration. Iterations is the number of batches or steps required for the full training or validation datasets to be processed. While the epoch value is how many times the whole dataset passes through the Neural Network. This value can be used to detect when overfitting occurs, which is when the number of epochs is increased until the accuracy metric with the validation data starts to decrease. Note, one epoch refers to one cycle through the full training dataset [46, 56, 57].

An advantage to having more than one epoch is that (especially for large training sets) it allows the Neural Network to use the previous data to update the weights and provide better model parameters so that the model is not biased towards the last few data points during training [46, 56, 57].

The batch size determines the number of iterations for a single epoch. Each batch contains several data samples. Once the batch has passed through the Neural Network, the predictions made are compared to the expected output. The model uses this error rate to update the weights to improve the model, to improve the prediction rate [46, 56, 57].

There are three standard batch methods used. When all training samples are used to create one batch, the learning algorithm is called batch gradient descent. When the batch is the size of one sample, the learning algorithm is called stochastic gradient descent. When the batch size is more than one sample and less than the size of the training dataset, the learning algorithm is called mini-batch gradient descent. The proposed models presented in chapter five used mini-batch gradient descent [46, 56, 57].

3.6.5 Dense Layer

A dense layer is when each neuron in the layer is connected to all the neurons in the previous layer. This is achieved by implementing the operation shown in equation 3.1 [46, 58]. Note that the parameters in equation 3.1 were discussed in the previous sections of chapter three. Finally, figure 3.14 shows an example of a densely connected layer.

$$\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias}) \quad \text{(Eq 3.1)}$$

where,

- The activation function is a scaling step to keep the neuron outputs within limits
- The kernel is a small size matrix for convolving the sample pixels of an image.
- Bias is effectively a neuron with a constant input of 1.

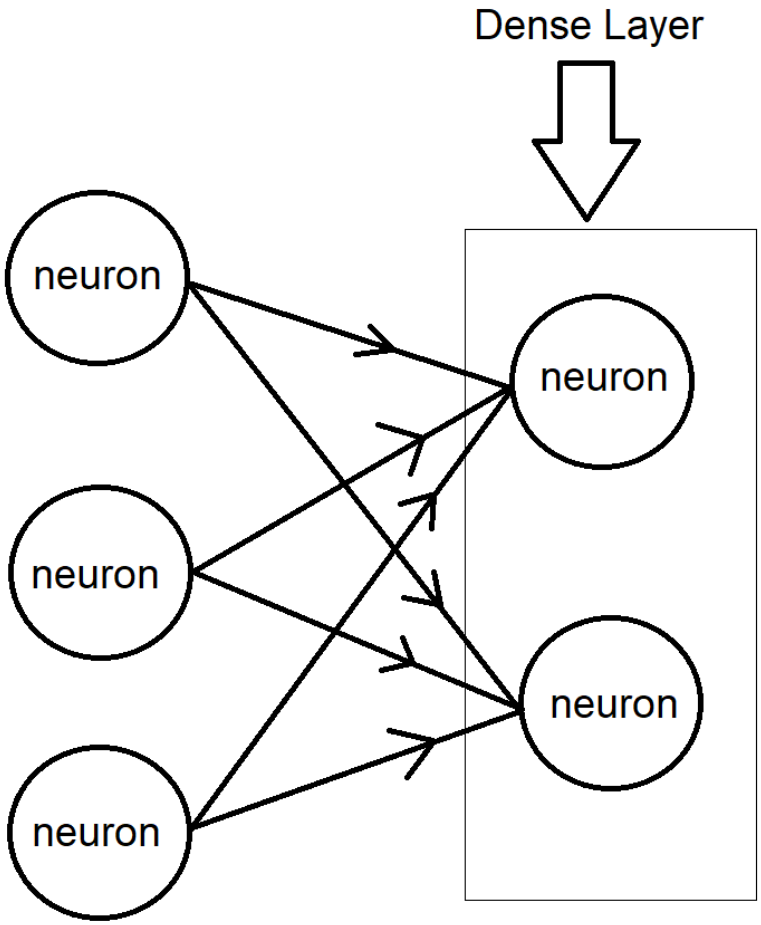

Fig. 3.14 Example of a dense layer.

3.6.6 Overfitting and Dropout

A common issue during training for Neural Networks is overfitting. This involves fitting the Neural Network model too closely to the training data. The model picks up on data points in training that are irrelevant for identifying that the data describing the label. For example, when identifying an image of a cat, the model may pick up the colour of the cat as an identifiable feature, and hence over fits [49, 55, 59, 60]. This typically occurs when using smaller datasets as there is less probability of the model correctly finding differences between labels (assuming supervised learning is being used). To address the issue of overfitting, a common technique is used, called dropout.

A Dropout layer will reduce the number of trainable parameters (discussed in chapter four) from the model to reduce the number of details it picks up. The intention is to thin the Neural Network of neurons, thereby only the essential details for identifying the label are left.

Dropout is achieved by cancelling a proportion of total connections between neurons. This is known as the dropout ratio. The dropped neuron connections mean they do not contribute to activating neurons during forward propagation as well as when updating the weights during backward propagation, hence less trainable parameters in the Neural Network. Therefore, overfitting will be minimised. This process is carried out during training of the Neural Network model, and the trainable parameters dropped (due to the cancelled connections from Dropout) are selected at random [46, 49, 55, 59, 60].

3.7 Fundamentals of Neural Networks

At its core, a Neural Network is constructed of neurons connected, either fully connected or partially connected, to each other. The layers between the input and output layers are called hidden layers. At the very minimum, there is an input layer and output layer with at least one hidden layer between them. If there is no layer between the input and output layers, then the relationship between the input and output is always linear. The number of hidden layers between the input and output and the number of neurons in each hidden layer is dependent on how complex the Neural Network needs to be. Neurons in one layer are connected to the neurons in the adjacent layers. If each neuron of a layer is connected to each neuron in the adjacent layer, that layer is considered fully connected (also known as a dense layer). Each connection between two neurons of adjacent layers has a weight assigned to that connection. As the input travels through the network, it is multiplied by the weights [35, 43, 61]. Each neuron has an activation function that defines the output of the neuron. Typically, the activation function is the same for each of the neurones in the same layer. The activation function is used to introduce non-linearity in the modelling capabilities of the network [46, 52].

The ultimate aim for any Neural Network is to learn the relationship between the inputs and outputs. The aim is achieved when all the weights are calculated correctly to reflect that relationship. The weights are calculated during training and validation. The calculation of the weights has two phases which are forward propagation and backward propagation, however not all Neural Networks have backward propagation. The following section of this chapter discusses the fundamentals of Neural Networks that allow for correctly calculating the weights [46].

3.7.1 Forward Propagation

The first step involves forward propagation. The training input data passes through the network, starting at the input layer until it reaches the output layer. When the data has passed through the layers and reaches the output layer, a prediction is made resulting in a label being assigned to the data, known as the calculated output. If there is no backward propagation, the weight values that produced the

minimum error between the calculated output and the ground truth of the output can be updated using any error minimisation methods. However, if there is a backward propagation phase, error minimisation and weight adjustments are carried out during the backward propagation phase instead [46].

3.7.2 Loss Function

The loss function depends on the error between the calculated output and the ground truth of the output. The loss function is also called Loss in this thesis. This step aims to minimise this error, ideally to zero. There are many mathematical approaches to minimise a loss function. The proposed work in this thesis used the gradient descent method to minimise the loss function. If the error is not zero, then the loss function can be further minimised by adjusting the weights more during backwards propagation [46].

While there are various types of loss functions, when the output of the Neural Network is categorical, categorical_crossentropy is the loss function used [46]. Note this was used in the proposed algorithm that produced the results presented in chapter five.

3.7.3 Backward Propagation

Backwards propagation starts from the output layer, moving backwards toward the input layer. The error between the calculated output and the ground truth of the output is used to calculate the outputs of each connected neuron in the backward adjacent layer. This process is propagated backwards through the network until the input layer is reached, and all the connected neuron outputs in each layer have been calculated. At this stage, the newly calculated outputs of the layers are used to update the weights on all the connections using error minimisation methods. For this research gradient descent was used [46].

3.7.4 Gradient Descent

Most loss function minimisation methods are based on a search for the trough using the function derivate. For this research, gradient descent is used. This is achieved by adjusting the weights using the learning rate, set by the optimizer in addition to the backpropagated error loss from step 3. Hence the changes in the weight values are minor unless learning rate decay is used. This process aims to find the absolute minimum of the error by using the loss functions gradient. When the error is minimised, the adjusted weights are then used as the new weights for the connected neurons. With the new weights forward propagation (step 1) occurs, and the cycle repeats [46].

Gradient descent algorithms can be iterative in that they start from a random point on a function and travel down its slope. The process ends when the trough has been reached. Figure 3.15 shows an example of a parabolic function representing a loss function equation to it. Applying gradient descent on the function y (figure 3.15) would mean the algorithm would need to calculate the value of x when y is minimum. An essential condition to consider is that when y is at the minimum, the gradient is zero. Hence the gradient descent algorithm works by calculating the gradient at random points until it reaches zero. The size of the steps it takes when selecting new points to test is known as the learning rate (this is discussed later in the chapter). The higher the learning rate, the bigger the step the algorithm takes, increasing the likelihood that the minimum point is skipped [46, 62].

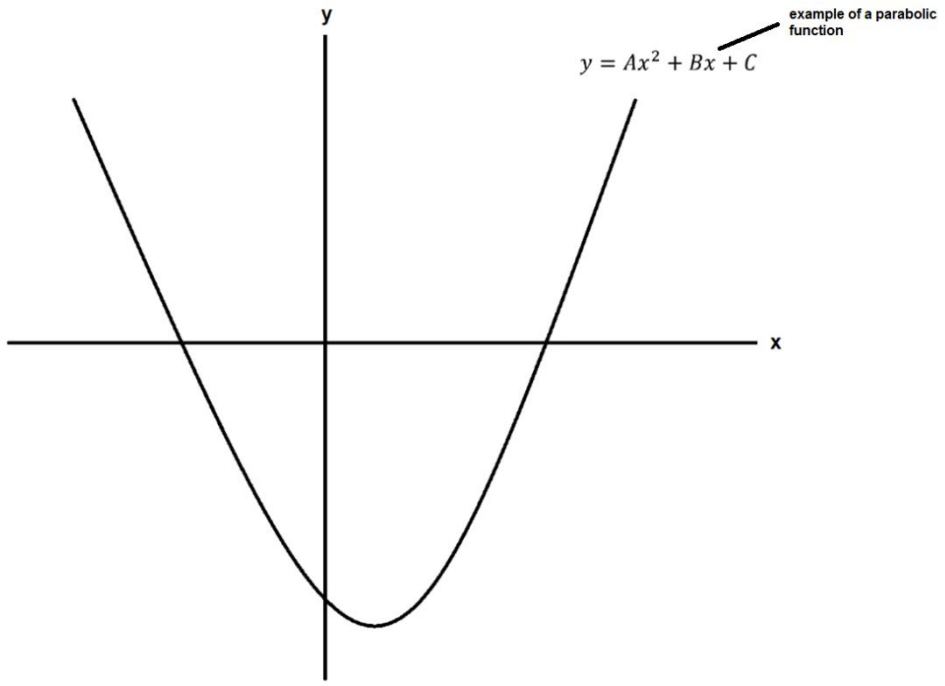

Fig. 3.15 Example of a parabolic function to illustrate gradient descent.

3.7.5 Optimizers

In general optimizers are algorithms used to adjust the weights by minimising the loss function using mathematical methods and learning rate. These parameters need to be continuously adjusted to minimise the loss function as much as possible. The gradient descent is the basis of many optimizers and one of the most common optimization algorithms in Machine Learning. The Keras [50] library provides several different optimizers for the user to implement in their Neural Network. The following are examples of optimizers available in Keras.

Stochastic Gradient Descent (SGD) optimises the gradient descent algorithm by introducing a random selection of data from the whole dataset. A more efficient process is to use SGD with mini-batch. This process requires the dataset to be broken down into small batches of data. Each batch is then passed through the Neural Network, where the gradient of its loss function is calculated and then can update the parameters of the Neural Network [46, 62].

Adagrad algorithms adapt the learning rate, depending on how frequently occurring the features are. For highly common features, the learning rate is reduced, and for rare features, the learning rate is increased [63, 64]. One downside to Adagrad algorithms is that they can lead to an aggressive decreasing learning rate, due to no limit in the accumulation of past gradients. The Adadelta optimizer solves this issue by restricting the window and therefore preventing the accumulated gradients reaching infinity. Instead, the result is a recent local estimate of the gradients [64].

Adaptive Moment Estimation (Adam) works similarly to Adagrad and Adadelta by storing past decaying gradients. This algorithm requires first-order gradients only and utilises minimal memory. Adam calculates different learning rates for various parameters. This is achieved by estimating the first and second moments of the gradients, which are stored as a running average [65, 66].

Nesterov Accelerated Adam Nadam (NAdam) is a variation of the Adam algorithm technique. The advantage NAdam has over Adam is that it uses Nesterov accelerated gradient (NAG) [66]. NAG is a first-order optimization method that improves the stability and convergence of decay gradients, such as the gradients Adam algorithms stores [46, 66, 67].

Using the correct optimiser is an essential part of any Neural Network. The learning process of a Neural Network can be viewed as a global optimization problem. Also, optimizers adjust parameters (weights and biases) in such a way, to reduce the loss function.

A parameter used by gradient descent optimizers is the learning rate. This parameter is used to multiply the magnitude of the gradient to determine the point on the curve (figure 3.15). To speed up this process, the learning rate decay method can be used. This involves starting with a high learning rate; hence big steps are initially taken. As the calculated gradient gets closer to 0, the learning rate value is reduced, so finer adjustments are made. This is important to reduce the risk of missing the trough of the curve in figure 3.15 [46, 56, 68].

3.8 Summary

This chapter discussed how Neural Networks function, specifically Convolutional Neural Networks (CNNs). Key aspects that can determine the accuracy of a Neural Network, during training and validation, including correctly adjusting the connection weights of neurons, were identified and will be discussed and evaluated further in chapter five.

A significant reason for the rise of deep learning is the computational power now available, which allows for larger dataset to be analysed. Kurzweil [69] stated that computational power is multiplied by a constant factor for each unit of time rather than just being added to incrementally. This means that computational power is increasing exponentially. More computational power also means faster processing and deeper Neural Networks (more layers) can be used.

The research allowed me to conclude that the most efficient type of Neural Network to use for classifying labels that use images are CNNs. Chapter four present the proposed algorithm based on the CNN approach and how training, validation (including evaluation) and testing were carried out. The outcome of the algorithm is my proposed models, which are discussed in chapter five.

3.9 References

[34] K. R. Awad M, *Efficient Learning Machines*. Berkeley: Apress, 2015.

[35] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," *arXiv preprint arXiv:1511.08458,* 2015.

[36] M. Newman, *Networks: An Introduction*. OUP Oxford, 2010.

[37] X. Zhu and A. B. Goldberg, "Introduction to semi-supervised learning," *Synthesis lectures on artificial intelligence and machine learning,* vol. 3, no. 1, pp. 1-130, 2009.

[38] M. Wiering and M. van Otterlo, *Reinforcement Learning: State-of-the-Art*. Berlin: Springer, 2014.

[39] L. Dormehl. "What is an Artificial Neural Network? Here's everything you need to know." DIGITAL TRENDS. https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/ (accessed August 2019).

[40] J. A. Hertz, *Introduction to the theory of neural computation*. 2018.

[41] A. Sears-Collins. "Artificial Feedforward Neural Network With Backpropagation From Scratch." Automatic Addison. https://automaticaddison.com/artificial-feedforward-neural-network-with-backpropagation-from-scratch/ (accessed December 2019).

[42] V. Valkov. "Making a Predictive Keyboard using Recurrent Neural Networks — TensorFlow for Hackers (Part V)." medium. https://medium.com/@curiousily/making-a-predictive-keyboard-using-recurrent-neural-networks-tensorflow-for-hackers-part-v-3f238d824218 (accessed January 2020).

[43] H. Nam and B. Han, "Learning multi-domain convolutional neural networks for visual tracking.," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pp. 4293-4302, 2016.

[44] G. E. Hinton, *A Practical Guide to Training Restricted Boltzmann Machines*. Berlin: Springer, 2012.

[45] C. Nicholson. "A Beginner's Guide to Restricted Boltzmann Machines (RBMs)." pathmind. https://pathmind.com/wiki/restricted-boltzmann-machine (accessed January 2020).

[46] J. Torres, *First Contact with deep learning: Practical Introduction with Keras*. Watch This Space, 2018.

[47] S. Saha. "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way." Towards Data Science. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (accessed June 2019).

[48] M. Yani, "Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail," in *Journal of Physics: Conference Series*, 2019, vol. 1201, no. 1: IOP Publishing, p. 012052.

[49] J. Brownlee, *Deep Learning for Computer Vision*

v1.7 ed.: Machine Learning Mastery, 2020.

[50] F. e. a. Chollet, "Keras," ed. GitHub repository: GitHub, 2015.

[51] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265-283.

[52] L. Vecci, F. Piazza, and A. Uncini, "Learning and approximation capabilities of adaptive spline activation function neural networks," *Neural Networks,* vol. 11, no. 2, pp. 259-270, 1998.

[53] T. Babs. "The Mathematics of Neural Networks." medium. https://medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05 (accessed December 2019).

[54] N. Kohl. "What is the role of the bias in neural networks?" Stack Overflow. https://stackoverflow.com/questions/2480650/what-is-the-role-of-the-bias-in-neural-networks (accessed May 2020).

[55] J. Brownlee, *Deep Learning with Python*, v1.18 ed.: Machine Learning Mastery, 2019.

[56] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1--learning rate, batch size, momentum, and weight decay," *arXiv preprint arXiv:1803.09820,* 2018.

[57] P. M. Radiuk, "Impact of training set batch size on the performance of Convolutional Neural Networks for diverse datasets," *Information Technology and Management Science,* vol. 20, no. 1, pp. 20-24, 2017.

[58] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700-4708.

[59] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research,* vol. 15, no. 1, pp. 1929-1958, 2014.

[60] S. Lawrence and C. L. Giles, "Overfitting and neural networks: conjugate gradient and backpropagation," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 2000, vol. 1: IEEE, pp. 114-119.

[61] Y. Zhao, J. Gao, and X. Yang, "A survey of Neural Network Ensembles," in *2005 International Conference on Neural Networks and Brain*, 2005, vol. 1: IEEE, pp. 438-442.

[62] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*: Springer, 2010, pp. 177-186.

[63] A. Lydia and S. Francis, "Adagrad-An Optimizer for Stochastic Gradient Descent," ed: May, 2019.

[64] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701,* 2012.

[65] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980,* 2014.

[66] A. Tato and R. Nkambou, "Improving adam optimizer," 2018.

[67] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," 2013: IEEE, pp. 8624-8628.

[68] G. Thimm, P. Moerland, and E. Fiesler, "The interchangeability of learning rate and gain in backpropagation neural networks," *Neural computation,* vol. 8, no. 2, pp. 451-460, 1996.

[69] R. Kurzweil, "The law of accelerating returns," in *Alan Turing: Life and legacy of a great thinker*: Springer, 2004, pp. 381-416.

Chapter 4: Proposed Algorithm

4.1 Introduction

The proposed algorithm architecture model was based on deep learning using convolutional layers followed by dense layers and was designed using Tensorflow Library from Google with the Keras library interface [50]. The proposed architecture uses five convolutions layers. Each Convolutional Layer used an activation function and has the same kernel size. A pooling layer immediately follows each convolutional layer. The pooling layer simplified the information from the convolutional layer, thereby producing a condensed version of the information. The final convolutional layer is followed by one dense layer connected to the output layer, which was also a dense layer with the number of outputs equal to the number of labels in the dataset. Chapter four discusses the proposed algorithm and how the model was compiled and how datasets [9-11] were split for training, validation, evaluation and testing.

The Deep Learning algorithm proposed in this thesis uses the Google Tensorflow library with the Keras library interface [50, 51]. Here I discuss the types of layers and fundamental steps regarding Keras layers. The Keras library is just an interface for the Deep Learning Library developed by Google Tensorflow.

4.2 Libraries

For the algorithm to work, it was required to import specific libraries (appendix 8.3.1). The NumPy library (np) is a scientific computing package for python [70, 71]. It provides users with an N-dimensional array object, along with the ability to integrate C/C++ code. The primary purpose of its use, in the proposed algorithm presented in this chapter, was to take advantage of the arrays it provides, which can be used to store different data types. Pandas is an open-source library first published in 2009 [72]. The library provides fast and efficient Data Frame objects allowing for data to be stored with indexing. In the proposed algorithm, the pandas library [72] was used to create 2- dimensional data structures to display information regarding Neural Network models and datasets. The library Sklearn.metrics [73] is a collection of machine learning tools, including statistical modelling. For my research, I used the libraries classification report and confusion matrix for evaluating the proposed models (discussed in chapter five). Similar to the Sklearn.metrics library, Matplotlib [74] also provides users with the ability to visualize data, both in static form and animated form. Hence, I used the library to generate image plots when testing the models.

The Tensorflow library [51] is an open-source platform for machine learning algorithms from Google. It produces data flow charts and graphs. For my research, Tensorflow was used to record, and display in graphs, the accuracy and loss measured during training and validation. Finally, the Keras library [50] is a high-level Application Programming Interface (API) of Tensorflow. The core data structure Keras provides are the layers and models that are used to design Neural Networks. The proposed algorithm uses the layers provided by Keras to form the architecture of the proposed models in chapter five.

4.3 Custom Functions

4.3.1 Class model_configuration

Among the custom functions, a class was created, model_Configuration (figure 4.1). This contained variables and directory paths that were repeatedly used in other functions. Hence, it made sense to put these in a class, so they were all in one place and easy to access. The commonly used variables were the epoch value, batch size, number of filters in the first layer, kernel size, number of neurons in the dense layer, number of labels and the learning rate. These variables determined the type of model that was compiled. The class also stored the dimensions of the images in the dataset as well as the total number of images available. For training to occur effectively, the images should have the same dimensions. Hence the adjusted dimensions were also stored. Finally, the stored directories were for the training, validation and testing datasets.

```
class model_Configuration:
    def __init__(self, train_dir, valid_dir,test_dir, epoch_value,
                 batch_size_value, num_first_layer_filters, kernel_size,
                 num_dense_layer_neurons, no_labels, learning_rate):
        # variables/parameters commonly used
        self.epochs = epoch_value
        self.batch_size = batch_size_value
        self.num_first_layer_filters = num_first_layer_filters
        self.kernel_size = kernel_size
        self.num_dense_layer_neurons = num_dense_layer_neurons
        self.no_labels = no_labels
        self.learning_rate = learning_rate

        # minimum and maximum dimensions of the images in the dataset.
        self.maxwidth = 0
        self.maxheight = 0
        self.minwidth = 1920
        self.minheight = 1080
        self.image_counter = 0

        # resoltion of images in data set
        self.img_width_adjust  = 480

        # resoltion of images in data set
        self.img_height_adjust = 360

        # directories for training, validation and test datasets
        self.train_data_dir = train_dir
        self.valid_data_dir = valid_dir
        self.test_data_dir = test_dir
```

Fig. 4.1 Python Class storing variables repeatedly used.

4.3.2 Image Dimensions function

In real-life applications, the dimensions of samples are not the same. It would be useful for the samples to be the same resolution. Otherwise, the maximum and minimum dimensions of the images would need to be calculated. For this, I used the function Image_Dimensions (figure 4.2). The function had one input which was the class, shown in figure 4.1. In the class function, the training dataset directory was stored which the function uses to walk through the directory (using the os library) to discover subdirectories and files in the given directory. If any file was of type .jpg or .jpeg (which were the image types used in the datasets used for this research), their dimensions were stored and recorded. A check was then made where if the current image had larger dimensions than the values currently stored

(in the class) regarding the maximum dimensions, then the stored variables in the class would be updated. The same process would occur when checking for minimum dimensions. This function only changes the variables stored in the class; hence it returns nothing.

```
def Image_Dimensions(input_class):
# stores image dimensions in the class called input_class
# uses the PIL library (Image module)
# uses os library
# 2 inputs: dataset path and input dictionary
    #image_directory = input_class.train_data_dir
    for subdir, dirs, files in os.walk(input_class.train_data_dir):
        for file in files:
          # check if the appropriate file types in path location
            if file.endswith(".jpg") or file.endswith(".jpeg"):

                input_class.image_counter +=  1

                # filename of current file
                current_filename = os.path.join(subdir, file)
                current_image = Image.open(current_filename)

                # store the width and height of the image
                current_width, current_height = current_image.size

                # check if current dimesions are larger or smaller
                # than min and max dimensions.
                if current_width < input_class.minwidth:
                    input_class.minwidth = current_width

                if current_width > input_class.maxwidth:
                    input_class.maxwidth = current_width

                if current_height < input_class.minheight:
                    input_class.minheight = current_height

                if current_height > input_class.maxheight:
                    input_class.maxheight = current_height
    return
```

Fig. 4.2 Function for determining image max/min dimensions

4.3.3 Dataset Setup

Before training can occur, the dataset needs to be setup. This process involves two main steps. Firstly, the dataset needs to be split into training and validation (if there is already no separate validation dataset). Secondly, the training dataset could be augmented to enhance sample diversity. However, if the sample space is ample, then image augmentation is unnecessary [49, 55]. The datasets used in my research were not diverse enough. Hence, I developed a function called dataset_setup (figure 4.3), which included image augmentation. The function was used to split the data into training, validation and testing. Note that the validation data was split from the training data directory, with the training receiving 80% of the data and validation receiving 20% of the data. The testing data (stored in test_samples) was also created and used the directory for the test samples. The training directory and the test directory were already formed from the dataset. dataset_setup took one input and returned three outputs. The input was the class called model_configuration (figure 4.1), while the (variable) outputs were the training, validation and test sample sets. It was these three variables that were contained the images that would be passed into the Neural Network model.

This function used two external functions called ImageDataGenerator and flow_from_directory. ImageDataGenerator is a Keras function [50] that alters images in the training dataset. Note the function did not affect the validation dataset. In this instance, the shear_range and zoom_range change the shear and zoom of the images, respectively. Another input for ImageDataGenerator is horizontal_flip which randomly flips some of the images in the horizontal axis. Finally, validation_split determined how much of the images are reserved for validation [49, 55].

The changes provided by ImageDataGenerator did not expand the training data the Neural Network model is exposed to during training. Instead, they ensure the training data, and the validation data are more different [49, 50, 55]. This is a crucial step as it makes it more likely the Neural Network will be tested on its intelligence and not on its memory. To test intelligence, the test data must not have been used during training and validation. Also, there needs to be significant differences between the images, so it is not relying on memory. This is why the images are altered and augmented, to increase the diversity between the training images and

the validation images. Achieving 100% training accuracy produces machine learning memory. This means the network is very good with the trained data only. Achieving 100% in evaluation or testing leads to machine learning intelligence. This means the model is good at identifying samples that it has never seen before. Machine learning intelligence is the desired outcome for any machine learning algorithm.

```
def dataset_setup (model_config):
    # ImageDataGenerator is used for augmentation of training data.
    # ImageDataGenerator takes batch of images and augments them before they
    # are sent though the Neural Network.
    # helps ensure bigger difference between validation data and training data
    # ImageDataGenerator does not affect validation data. only determines split
    # not all options necessary. only which would be considered realistic.
    # horizontal flip is realistic. vertical flip not realistic
    # validation_split sets how much of training data should be for validation
    sample_data_gen = ImageDataGenerator(rescale=1. /255, shear_range=0.2,
                                         zoom_range=0.2, horizontal_flip=True,
                                        validation_split=0.2)
    # create training dataset.
    # shuffle randomises images
    # colour mode sets colour of images.
    # class mode if multiple outputs use categorical
    # subset important when spliting validation data from same set.
    train_samples = sample_data_gen.flow_from_directory(
                    model config.train data dir,
                    target_size=(model_config.img_width_adjust,
                                 model_config.img_height_adjust),
                    shuffle=True,
                    color mode = "rgb",
                    batch size=model config.batch size,
                    class mode='categorical',
                    subset='training')
    # create validation generator
    # shuffle important when distribution of data is in order
    # splitting data means validation gets last 10% not ideal
    valid_samples = sample_data_gen.flow_from_directory(
                     model_config.valid_data_dir,
                     target_size=(model_config.img_width_adjust,
                                  model_config.img_height_adjust),
                     shuffle=True,
                     color_mode = "rgb",
                     batch_size=model_config.batch_size,
                     class_mode='categorical',
                     subset='validation')
    # create test dataset
    test_samples = sample_data_gen.flow_from_directory(
                   model_config.test_data_dir,
                   target_size=(model_config.img_width_adjust,
                                model_config.img_height_adjust),
                                        batch_size=model_config.batch_size,
                                        class_mode='categorical')
    # Note uses training dataflow generator
    return train_samples, valid_samples, test_samples
```

Fig. 4.3 Setup data function

With the altered data produced, the function flow_from_directory was used to generate batches of the augmented data. The function took seven inputs which were the directory of the data which it wants to create (training data directory for train data), the target_size, shuffle (set to true), colour mode, batch_size, class mode and subset.

The target_size provided adjusted dimensions for the images, so they were all the same size when passing through the model. The batch_size was provided by the input class, which determined how many images would pass into the model in one go. The class_mode described the type of dataset being used. The general rule is when the dataset is categorised into multiple labels, then class mode categorical should be used [50]. The AUC Distracted Driver Dataset [10, 11] that I trained the proposed model on was a categorical dataset. Hence the class mode categorical was used. Finally, the subset was which type of data to produce. This is only used when the ImageDataGenerator function [50] uses the validation split variable. When used, it is vital to use the variable subtype to distinguish between the training and validation datasets.

4.3.4 Compile Model

Before training can occur, the Neural Network model needed to be compiled (figure 4.4). The parameters that determined the shape of the model were provided by the class model_configuration (figure 4.1). Hence it was the input variable to the compile_model function, shown in figure 4.4. This function returned one output which was the compiled and untrained model. The input class was specifically used to create the convolutional layers, as well as the input, output and dense layers. The returned output was the compiled, untrained model. The proposed models, discussed in chapter five, were designed with five convolutional layers and a fully connected layer, which was located between the input layer and a dense output layer. The function Conv2D [50] was used to produce the convolutional layers, with activation function ReLU. Each convolutional layer had one Conv2D function and one MaxPooling2D function [50]. The Conv2D function was the constructor that adds the filters that were to be learned.

Each Conv2D function was immediately followed by a MaxPooling2D function which was used to summarize an image, by producing a smaller representation

which allowed assumptions to be made of the features in the image [75]. This process reduced the number of trainable parameters, thereby saving computational power. Also, by lowering the trainable parameters, the likelihood of overfitting was reduced. When a Neural Network model has too many trainable parameters, insignificant features are picked up. These features identify parts of the image, which are irrelevant to the image's label. For example, consider an image showing an operator using a radio, and the label is operating the radio. If the model picks up the colour of the operators' clothes as a feature to identify the label, this is considered overfitting [59, 60].

Once the number of convolutional layers was determined, it is necessary to connect them to the output layer, which the number of neurons in it is determined by the number of labels in the dataset. However, the output layer used was one dimensional, and so to successfully connect to a 3-dimensional convolutional layer, a flatten layer was used. This layer connected to a dense layer, which is when each of its neurons is connected to all the neurons in the previous layer [58]. Finally, the dense layer was connected to the output layer.

There are two main approaches to the adaptive approach that I investigated. The first approach was to use a compiled model that had epoch set to 1, and the model itself was placed in a loop. The number of iterations of the loop is the number of epochs. This allowed me to adjust any parameters, between iterations, which would have a positive effect on the training and validation accuracy. This approach allowed me to monitor the error and performance of each label. Hence this allowed for adaptive learning for adjusting the model parameters of the learning outcome. The second approach was to update the weights at the end of the forward phase using a learning rate model. The weights were further adjusted at the end of backward propagation. Both approaches are discussed further in chapter six, future work.

```
def compile_model(input_class):
# function builds Neural Network model.
# takes 6 inputs and returns the compiled model.
# input_class has data required by input layer.
# 5 layers used.
    # input layer. dimensions of images given as well as if image is rgb
    # or grayscale.
    # if rgb then 3 is the input, if grayscale then 1 is the input.
    Input_layer = Input(shape=(input_class.img_width_adjust,
                               input_class.img_height_adjust, 3), name="input")
    # each connvolutional layer followed immediately by pooling layer.
    # convolutional layer connected to its pooling layer.
    # each pooling layer connected to the next convolutional layer.
    conv1 = Conv2D(input_class.num_first_layer_filters, kernel_size =
                  (input_class.kernel_size, input_class.kernel_size),
                   activation="relu", name="conv_1")(Input_layer)
    pool1 = MaxPooling2D(pool_size=(2, 2), name="pool_1")(conv1)
    conv2 = Conv2D(input_class.num_first_layer_filters // 2, kernel_size =
                  (input class.kernel size, input class.kernel size),
                   activation="relu", name="conv_2")(pool1)
    pool2 = MaxPooling2D(pool size=(2, 2), name="pool 2")(conv2)
    conv3 = Conv2D(input_class.num_first_layer_filters // 4, kernel_size =
                  (input_class.kernel_size, input_class.kernel_size),
                   activation="relu", name="conv_3")(pool2)
    pool3 = MaxPooling2D(pool size=(2, 2), name="pool 3")(conv3)
    conv4 = Conv2D(input_class.num_first_layer_filters // 8, kernel_size =
                  (input_class.kernel_size, input_class.kernel_size),
                   activation="relu", name="conv_4")(pool3)
    pool4 = MaxPooling2D(pool_size=(2, 2), name="pool_4")(conv4)
    conv5 = Conv2D(input_class.num_first_layer_filters // 16, kernel_size =
                  (input_class.kernel_size, input_class.kernel_size),
                   activation="relu", name="conv_5")(pool4)
    pool5 = MaxPooling2D(pool_size=(2, 2), name="pool_5")(conv5)
    # Fully Connected Layer Dense. Flatten function required before Dense.
    flatten = Flatten()(pool5)
    fc1 = Dense(input_class.num_dense_layer_neurons, activation="relu",
                name="fc_1")(flatten)
    # output layer also dense. has 10 neurons whicha are the labels.
    output_layer = Dense(input_class.no_labels, activation="softmax",
                         name="softmax-output")(fc1)
    # Finalize and compile
    # Loss different depending on number of labels at the output.
    # If 2 labels then binary_crossentropy is used.
    # If more than 2 then categorical_crossentropy is used.
    compiled_model = Model(inputs=Input_layer, outputs=output_layer)
    compiled_model.compile(optimizer=Adam(input_class.learning_rate),
                           loss='categorical_crossentropy',
                           metrics=["accuracy"])
    return compiled_model
```

Fig. 4.4 Compile the Neural Network model

4.3.5 Dataset and Model Information

An essential aspect of any Neural Network that uses labelled data is how the dataset is structured. The datasets used to produce my results and test my Neural Network were structured by having each label in a separate folder. Therefore, I developed a function called dataset_model_information (figure 4.5), which counts the number of files in a specified path. I used it to count the number of images in each label and store the counts into a list called directoy_count. This was possible because each label had its own directory.

```
def dataset_model_information(input_class, compiled_model):
# lists the directories in the given path and provides number of files in
# each subdirectory
    # empty list formed.
    # this will be the output after calling this function.
    directoy_count = []
    for subdir, dirs, files in os.walk(input_class.train_data_dir):
        # gives the number of files in the current directory and
        # stores them in input_class.image_counter.
        input_class.image_counter = len(files)
        # current directory (or sub directory stored in dirname)
        dirname = subdir
        # list appended with each set of location and file count.
        # output directoy_count is returned after calling this function.
        directoy_count.append((dirname, input_class.image_counter))


    categoryInfo=pd.DataFrame(directoy_count,columns=['Category','Count'])
    categoryInfo = categoryInfo.sort_values(by=['Category'])

    print(categoryInfo.to_string(index=False))
    print("Minimum Width:\t", input_class.minwidth,
          "\tMinimum Height:", input_class.minheight)
    print("Maximum Width:\t", input_class.maxwidth,
          "\tMaximum Height:", input_class.maxheight)
    print("Image Count:\t", input_class.image_counter)
    print(compiled_model.summary())
    return
```

Fig. 4.5 Function for displaying model information

The function takes two inputs and has no outputs. Instead, it prints data to the built-in command window/console in the users chosen editor. The inputs for this function are an input class and a compiled (untrained) model. The class is the model_Configuration class that was used to store specific data (figure 4.1). The

function created a list containing each subdirectory and their respective image count. The pandas (pd) data structure [72] was used to produce a 2-column table containing the path of each subdirectory (each label of the dataset) and their respective file count (number of images). When displayed, they were sorted alphabetically with respect to the path (category column). As an output, the function printed out the maximum and minimum dimensions, as well as the image count. Lastly, the function showed a summary of the compiled (untrained) model. This function was unnecessary to run the algorithm, as it only printed information regarding the model and dataset. However, the presented information was useful to check the compiled models shape and important details about the dataset.

Figure 4.6 shows an example summary of a compiled model. When using the Keras function .summary [50], three pieces of information are provided; the type of layer, its shape and the number of parameters each layer has. Lastly, at the bottom of the figure is the total number of trainable parameters. Where the output shape shows three values, the first 2 are the dimensions of the images, with the third showing the number of outputs that the layer has. Note as more layers are added, the image dimensions are reduced. The number of parameters for each convolutional layer (note max-pooling layers do not have parameters), were calculated using equation 4.1, shown below [76, 77].

$$no.of\ param_i = layer\_outputs_i \times (layer\_outputs_{i-1} \times kernel\ area + 1) \quad \text{(Eq 4.1)}$$

Where i is the layer which the number of parameters is being calculated for. The Kernel area is calculated from the dimensions of the Kernel size. For the model proposed, the kernel dimensions used were 3x3; hence the Kernel area was 9. The number of parameters a layer has is determined by its output and the output of the previous layer. Looking at figure 4.4, the number of first layer filters (num_first_layer_filters) is used to calculate the number of outputs the layer has (Equation 4.2) [76, 77].

$$number\ of\ outputs_n = \left\lfloor \frac{num\_first\_layer\_filters}{\frac{2^n}{2}} \right\rfloor \quad \text{(Eq 4.2)}$$

Where, n = 1 means first convolutional layer (conv_1).

Hence as more layers are added, the number of outputs the latter convolutional layers have decreases, along with the number of parameters. The exception is the first convolutional layer, where the number of parameters is less than the second layer. This is due to it being attached to the input layer, which only has three outputs [76, 77].

```
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           (None, 480, 360, 3)       0
_________________________________________________________________
conv_1 (Conv2D)              (None, 478, 358, 128)     3584
_________________________________________________________________
pool_1 (MaxPooling2D)        (None, 239, 179, 128)     0
_________________________________________________________________
conv_2 (Conv2D)              (None, 237, 177, 64)      73792
_________________________________________________________________
pool_2 (MaxPooling2D)        (None, 118, 88, 64)       0
_________________________________________________________________
conv_3 (Conv2D)              (None, 116, 86, 32)       18464
_________________________________________________________________
pool_3 (MaxPooling2D)        (None, 58, 43, 32)        0
_________________________________________________________________
conv_4 (Conv2D)              (None, 56, 41, 16)        4624
_________________________________________________________________
pool_4 (MaxPooling2D)        (None, 28, 20, 16)        0
_________________________________________________________________
conv_5 (Conv2D)              (None, 26, 18, 8)         1160
_________________________________________________________________
pool_5 (MaxPooling2D)        (None, 13, 9, 8)          0
_________________________________________________________________
flatten_7 (Flatten)          (None, 936)               0
_________________________________________________________________
fc_1 (Dense)                 (None, 1024)              959488
_________________________________________________________________
softmax-output (Dense)       (None, 10)                10250
=================================================================
Total params: 1,071,362
Trainable params: 1,071,362
Non-trainable params: 0
```

Fig. 4.6 Example of a summary of a compiled model.

4.4 Training and Validation

The training code started by setting the variables that would determine the shape and parameters of the Neural Network (figure 4.7). These were the epoch, the batch_size, learning_rate, num_first_layer_filters, kernel_size, no_labels and num_dense_layer_neurons. The epoch variable determined the number of times the model will pass the entire training dataset through it. The learning_rate variable determined how much the weights changed with each epoch. The smaller the learning rate value that was set, the smaller the change in the weight value after each epoch.

Similarly, the larger the learning rate value provided, the bigger the change in the weight values after each epoch. num_first_layer_filters determined the number of outputs for each layer (figure 4.4), hence also determining the number of trainable parameters. The kernel size determined the dimensions of the kernel, which was an operator that was applied to the entire image [35, 78]. The kernel was a 2D matrix of numbers that passed over the image and multiplied its matrix value with the corresponding pixel value. The results were summed up, and the new value was used to represent the grid cell (which is the same size as the kernel dimensions), thereby simplifying the image. Variable num_dense_layer_neurons determined the number of neurons in the dense layer. Note, a dense layer is when each neuron in one layer is connected to all the neurons in the previous layer [58]. Finally, the no_labels variable was the number of labels the dataset uses. For the AUC Distracted Driver Dataset [10, 11], which recorded driver distraction, there were ten labels.

Once the variables were assigned values, variables SAVED_WEIGHTS_NAME and SAVED_MODEL_NAME were used to name the weights and model files, respectively. These files were required for evaluating the model to ensure the best weights were loaded into the model, and not always the last weights.

The variable model_info was used to store the class called model_Configuration (figure 4.1). This class required ten inputs, which included the variables at the beginning of the code in figure 4.7. The TRAIN_DATA_DIRECTORY variable stored the directory where the training data was located, the

VALID_DATA_DIRECTORY stored the location for the validation data, and TEST_DATA_DIRECTORY stored the directory of the test data.

With this data produced, the dataset can was split into training data and validation data, which were stored in train_generator and valid_generator variables, respectively. Using the custom dataset_setup function I produced. This function split the data in the training folder, for the AUC dataset [10, 11] into training data and validation data. The split was 80% for training data and 20% for validation data. For dataset_setup to function, it needed one input, the class model_info. The required variables stored in this class included the directories for the training data, validation data and test data. Similar splits were done for the UTA-RLDD dataset [9].

Once the data had been correctly split and stored in the outputs for the dataset_setup function (figure 4.3), the model was built using the compile_model function (figure 4.8) from the Keras library [50]. This function had one input and returned one output. In this case, the returned output, the compiled model, was stored in the variable my_compiled_model. The input was the class model_info, which was needed for its variables kernel_size, num_first_layer_filters, num_dense_layer_neurons, the no_labels and the learning_rate. Once the model was compiled, the dataset_model_information function was executed. This took two inputs, which were the class model_info and the compiled model. This function displayed the built model, showing the shape and number of parameters of each layer (an example is shown in figure 4.6).

```
np.random.seed(42)

# name model and its weights saved under
model_version = "model_name"

epoch = 5
batch_size = 32
learning_rate = 0.001
num_first_layer_filters=128
kernel_size=3
num_dense_layer_neurons=1024
no_labels = 10

TRAIN_DATA_DIRECTORY =
'/content/drive/My Drive/Colab Notebooks/AUC\v1_cam1_no_split/train'
VALID_DATA_DIRECTORY =
'/content/drive/My Drive/Colab Notebooks/AUC\v1_cam1_no_split/train'
TEST_DATA_DIRECTORY =
'/content/drive/My Drive/Colab Notebooks/AUC\v1_cam1_no_split/train'

# naming the weights and models with the variable (and its assigned value)
# that is be investigated.
SAVED_WEIGHTS_NAME = "myweights_" + model_version + ".h5"
SAVED_MODEL_NAME   = "mymodel_" + model_version + ".json"

# model_configuration called
model_info = model_Configuration(TRAIN_DATA_DIRECTORY,
                                 VALID_DATA_DIRECTORY,
                                 TEST_DATA_DIRECTORY,
                                 epoch,
                                 batch_size,
                                 num_first_layer_filters,
                                 kernel_size,
                                 num_dense_layer_neurons,
                                 no_labels,
                                 learning_rate)
# dataset_setup called
# data generated. traiing and validation split 80% and 20% respectively.
train_generator,valid_generator,test_generator = dataset_setup(model_info)
```

Fig. 4.7 Setup and Generate data

```
# compile model
my_compiled_model = build_model(model_info)

# display model summary and dataset information
dataset_model_information(model_info, my_compiled_model)

# serialize model to JSON
mysaved_model = my_compiled_model.to_json()
with open(SAVED_MODEL_NAME, "w") as json_file:
    json_file.write(mysaved_model)

# save location for model.
save_dir_tf = '/content/drive/My Drive/Colab Notebooks'
              + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
              + model_version

# At the beginning  save the  first weights
# Only save weights if the current val_acc is better than the val_acc
# saved file weights
checkpoint = ModelCheckpoint(SAVED_WEIGHTS_NAME, monitor='val_acc',
                             verbose=1,
                             save_best_only=True,
                             mode='max')
```

Fig. 4.8 Compile and save the model

With the model compiled it was saved to a .json file. Finally, With the model built and the data split correctly, training the model was the next stage, shown in figure 4.9. While training the model, Tensorboard was used for logging the accuracies and losses achieved with each batch. The final model output only stored the data that produced the highest accuracy. The function ModelCheckpoint (figure 4.8), a callback function provided by Tensorflow [51] was used during training (figure 4.9). It saved the best validation values produced during training and validation for each epoch.

fit_generator is the Keras function used to train models (figure 4.9). Hence, the compiled model from figure 4.8, was trained using fit_generator. Fit_generator takes seven inputs. Train_generator was the data assigned for training, while steps_per_epoch and validation_steps determined the number of steps each epoch required for the model to have been trained on the whole training data once. The epochs variable is the number of times the training will go through the entire training dataset before completion. The variable validation_data was the data assigned for validation. The variable verbose was used to determine how the data

is displayed when printed. The verbose value was set to 1, which shows a progress bar as each batch_size goes through the model. Verbose had no impact on the results and could have just as easily shown nothing (set to zero). The callbacks input used the tbCallback and checkpoint variables log statistics of the model while training occurred [51]. This allowed Tensorboard to provide graphs of the accuracy and loss during training and validation of the Neural Network.

```
# the checkpoint callback is used for tracking the validation accuracy to
# save the best weights.
tbCallback = TensorBoard(log_dir=save_dir_tf,
                         histogram_freq=0,
                         write_graph=True,
                         write_images=True)

# fit_generator trains the compiled model.
# checkpoint for saving weights  that give the best validation accuracy.
# saved weights are saved in a .h5 file.
# tb callback for saving training losses and accuracies
trained_model.fit_generator(train_generator,
                                steps_per_epoch = train_generator.samples
                                                  //model_info.batch_size,
                                epochs = model_info.epochs,
                                validation_data = valid_generator,
                                validation_steps= valid_generator.samples
                                                  //model_info.batch_size,
                                verbose=1,
                                callbacks=[checkpoint, tbCallback] )
```

Fig. 4.9 Training

During training, the model and the weights that produced the highest accuracy were saved into two files. The model was saved to a .json file (figure 4.10), while the weights are saved to a .h5 file (figure 4.10) using the modelcheckpoint function [51]. The model was saved using the with open function to be able to write to the file the complied model from my_compiled_model.fit_generator function (figure 4.10).

4.5 Evaluation

Once training was complete, the confusion matrix and classification report were developed using the validation dataset figure (4.10). To achieve this, the model and its respective weights were loaded into a new variable called trained_model (figure 4.9). The sklkearn library [73] versions of the confusion matrices and classification reports were used to produce a prediction of the accuracy of the trained model on data it had not seen before. Hence tested the intelligence of the model and not its memory. Chapter five explains further what the classification report and confusion matrices are and how they were produced.

```
#valid_generator is the validation dataset created with the function: data
set_setup
#valid_generator.samples is from validation_steps in the function: fit_gen
erator
# batch_size - variable loacted at begining.
scores= my_compiled_model.evaluate_generator(valid_generator,
                                             steps=valid_generator.samples
                                              // batch_size)
print("Evaluation of compiled model with weights loaded.")
print("Loss: " + str(scores[0]) + " Accuracy: " + str(scores[1]))

# Scikit-learn version of the Confution Matrix and Classification Report
# floor division -> "num_of_test_samples // config.batch_size + 1"
# batch size for training and validation is the same value  = 16.
# resetting generator. important if valid_generator used for prediction
valid_generator.reset()

prediction_matrix = my_compiled_model.predict_generator(valid_generator,
                                            steps=valid_generator.samples
                                                        // batch_size +1)
# np.argmax gives the maximum indicies in prediction_matrix
max_pred_value = np.argmax(prediction_matrix, axis=1)

#calculate and print confusion matrix and classifaction report.
print(' Scikit-learn Confusion Matrix')
print(confusion_matrix(valid_generator.classes, max_pred_value))
print('Scikit-learn Classification Report')
print(classification_report(valid_generator.classes, max_pred_value))
```

Fig. 4.10 Confusion matrix and classification report

4.6 Testing

Before testing the trained and validated model, some setup was required, which can be seen in figure 4.11. The path to the test images the trained model and the location to save the results were required. Finally, the labels were placed in a dictionary (figure 4.12) which could be easily accessed when assigning the label to the test data (which was unlabelled and unseen by the trained model).

```
# testing intelligence of model. Hasn't seen these images
# testing file.
IMAGES_PATH = '../_DataSet_/AUC\v1_cam1_no_split/test/'

# Load model and set path directory to save result.
MODEL_PATH = 'model_name.h5'
SAVE_RESULTS_DIR = './result'
```

Fig. 4.11 Testing code setup

```
# dictionary labels made of key and value.
# These labels are for AUC dataset
MAPPING_DICTIONARY = {0: "Safe Driving",
                      1: "Texting Right",
                      2: "Talking on the Phone - Right",
                      3: "Texting Left",
                      4: "Talking on the Phone - Left",
                      5: "Operating the Radio",
                      6: "Drinking",
                      7: "Reaching Behind",
                      8: "Hair and Makeup",
                      9: "Talking to Passenger"}
```

Fig. 4.12 Dictionary of labels

Before an image could be tested, it needed to be resized using the Resize_Image_for_CNN function (figure 4.13). This function had three inputs, which were the input image and width and height dimension to resize the images to. The resizing function came from the OpenCV library (resize), and the function

expand_dims came from the NumPy library [70]. The resized image is then returned.

```
# resize images
def Resize_Image_for_CNN(image, width, height):
    resized_image = cv2.resize(image, (width, height))
    resized_image = resized_image.astype('float32')
    resized_image /= 255.

    # expand the dimensions of the image
    resized_image = np.expand_dims(resized_image, axis=0)
    return resized_image
```

Fig. 4.13 Resize an image.

Testing the trained model involved in inputting the resized image into the model and predicting the correct label. Figure 4.14 shows how this is achieved. The Keras predict function [50] was used to determine which label is the most likely correct one to describe the unlabelled image. The most likely label is picked from the dictionary (figure 4.12) bypassing the winner_class variable as the key for the dictionary. The key provides the correct value, which in this case is a string describing the action the driver is taking. The variable winner_class contains an integer given when attempting to predict the image's correct label. To speed the process of testing up, I created a plot of figures to show a batch of tests quickly. Figure 4.14 shows the plot set to have a size of 5x5.

```
# variables initialized
rows = 5; cols = 5; f = []; idx = 1
for (dirpath, dirnames, filenames) in walk(IMAGES_PATH):
    f.extend(filenames)
    break
# load the model by using path of the model
cnn_model = load_model(MODEL_PATH)
# create list of images and store in imageFileNameList
imageFileNameList = list()
imageNameList = [ f for f in os.listdir(IMAGES_PATH) ]
imageFileNameList += [os.path.join(IMAGES_PATH, file)
                      for file in imageNameList]
# create plot of figure size rows x cols
plt.figure(figsize=(rows, cols))
# test model on images one at a time. loops through imageFileNameList
for imageFileName in imageFileNameList :
    image = cv2.imread(imageFileName)
    cv2.imshow('out', image)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    cv2.imshow('out', image)
    input_cnn = make_image_read_for_cnn(image)
    # winner_class assigns image a label from dictionary
    cnn_output = cnn_model.predict(input_cnn)
    winner_class = cnn_output[0].argmax()
    # print label on image
    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(image, MAPPING_DICTIONARY[winner_class],
                (10, 30),font, 1, (0, 0, 255), 1, cv2.LINE_AA)
    cv2.imwrite('test.png', image)
    cv2.imshow('out', image)
    # Only executes if there is a free plot space to inset image
    # if no free plot space then new plot is created.
    if idx <= 25:
        full_path_name =  'test.png'
        plt.subplot(5, 5, idx)
        idx = idx + 1 ;
        img = plt.imread(full_path_name)
        plt.imshow(img)
        plt.tight_layout()
cv2.waitKey(0) # waits for user input
```

Fig. 4.14 Main code for testing

4.7 Summary

The Neural Network model was designed as a convolutional network model. The input layer was followed by five convolutional layers, with each having their own max-pooling layer. The final layer has a dense connection with the fifth convolutional layer. The model was trained, validated and tested using the AUC dataset [10, 11] and the UTA-RLDD dataset [9]. During training TensorBoard, TensorFlow's visualisation toolkit [51], was used to record the accuracy and loss as the training data passed through the model. Classification reports and confusion matrices were used to make predictions on how accurate the model will be at assigning the correct label.

With the proposed algorithm explained, the proposed models developed from this algorithm are presented and evaluated in chapter five.

4.8 References

[70] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: a structure for efficient numerical computation," *Computing in Science & Engineering,* vol. 13, no. 2, pp. 22-30, 2011.

[71] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006.

[72] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, 2010, vol. 445: Austin, TX, pp. 51-56.

[73] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *the Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[74] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in science & engineering,* vol. 9, no. 3, pp. 90-95, 2007.

[75] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks,* vol. 61, pp. 85-117, 2015.

[76] R. Vasudev. "Understanding and Calculating the number of Parameters in Convolution Neural Networks (CNNs)." https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-fc88790d530d (accessed 10 December 2019).

[77] Y. Zhang. "Number of Parameters in Dense and Convolutional Layers in Neural Networks." medium. https://medium.com/@zhang_yang/number-of-parameters-in-dense-and-convolutional-neural-networks-34b54c2ec349 (accessed August 2019).

[78] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285,* 2016.

Chapter 5: Proposed CNN Models.

5.1 Introduction

Chapter four discussed the proposed algorithm, which was designed based on a Convolutional Neural Network model. The algorithm was designed using the Keras library [50] in the python language.

There are three stages to developing a Neural Network model. Once the model was created, it must be trained, validated and finally tested. The validation dataset was derived by splitting up the training dataset. The split was 80% for training and 20% for validation. Tensorboard was used to create the graphs for the accuracy and loss values for each trained model.

During training, the model is learning the differences between the labels. During validation, the model is being tested on data unseen during training. The validation accuracy and loss, which I recorded during simulation runs, (along with training accuracy and loss), is a good indication of whether the model under fitted, overfitted or ended up with a good fit, regarding the data it is modelling itself on. The difference between the validation accuracy and training accuracy is a good indicator of this. If validation is much lower than training, then it is likely overfitting occurred. A low training accuracy indicates underfitting, and lastly, when the validation and training accuracies are similar, it can indicate the model ended up with a good fit.

Finally, testing the data involved looking at unlabelled images which it did not see during training or validation. This testing technique examined the intelligence of the Neural Network. If testing involved looking at images it and seen during training, then the Neural Network's memory would be tested instead. Another critical aspect of testing the model’s intelligence was that each piece of data needs to be sufficiently different, to ensure it is not relying on its memory.

Chapter five discusses a range of different investigations that were carried out using two databases. The effects of specific parameters were looked at, as well as how the accuracy changed depending on the structure of the datasets. Finally, the model that produced the highest level of accuracy for each dataset is presented in this chapter.

5.2 Datasets

The proposed algorithm was trained and validated on two datasets to determinate the effectiveness. Both the training and validation were done using labelled data (supervised learning). During training, the Neural Network required labelled data to learn the differences between the labels. Validation needed labelled data so an evaluation on a labelled image could be determined between the prediction of the label the model makes and the real (true) label assigned to the image. The effectiveness of the algorithm was determined in terms of accuracy and loss during training and validation. Using two different datasets, the algorithm could be tested, and its operation observed under different conditions (due to the differences in the datasets).

The first dataset used was the AUC dataset [10, 11], which the developers of it were inspired by the State Farm distracted driver detection dataset [79], used In a Kaggle competition in 2016. The AUC dataset consisted of 10 labels, which are all related to distractive behaviour, from using the radio to talking on the phone. The training dataset from the AUC dataset contained 17,308 labelled images. From this, 20% were assigned to the validation dataset, with the rest (80%) assigned to training. The testing dataset contained 1123 labelled images. However, not all were used during the testing phase. Instead, the evaluation of the model was used to determine how well it would do during testing.

The second dataset used was the University of Texas at Arlington Real-Life Drowsiness Dataset (UTA-RLDD) [9]. The dataset consists of 180 RGB videos, from sixty participants, each lasting roughly 10 minutes. Each participant recorded three videos which were sorted into three labels: alert, low vigilant and drowsy. The data these videos provided range from easily visible cases of fatigue to very subtle cases of fatigue. Before this database could be used, images were extracted from each video every 150 frames. This resulted in approximately 100 to 120 images per video. The videos were recorded in different aspect ratios and resolutions; hence when the images were produced, they mirrored this. Due to the data being uninformed, initial testing found the models having difficulty with training. It was discovered that the main issue was with the low vigilant label. The data in this label did not differ enough from either of the other labels and hence was the hardest to identify correctly. Hence, I decided to look at only the extreme labels of alert and

drowsy. As such, the investigations discussed in this chapter, regarding the fatigue dataset, were carried out on those two labels.

5.3 Confusion Matrix

A Confusion matrix, using the Scikit-learn library [73], shows predicted values along the column and the actual values as the rows [80]. Each column and row represent the labels the dataset used. For the AUC driver distraction database [10, 11] the training labels were assigned, ordered from top to bottom, and left to right, Safe Driving, Texting Right, Talking on the Phone - Right, Texting Left, Talking on the Phone - Left, Operating The Radio, Drinking, Reaching Behind, Hair and Makeup and Talking to Passenger. Likewise, the UTA-RLDD dataset [9] had three labels that were assigned to the rows and columns. The order of the labels from top to down and left to right were alert, vigilant and drowsy. Table 5.1 shows an example of a confusion matrix. The expectation is the combinations of labels with the highest success rate would be when they are the same (highlighted numbers in table 5.1). for example, label 1 was predicted to be true (as label 1) 50 times and was predicted incorrectly 13 times (summation of the true labels, 2 to 5, that were predicted to be label 1).

Table. 5.1 Example confusion matrix

		PREDICTED				
		label 1	**label 2**	**label 3**	**label 4**	**label 5**
TRUE	**label 1**	50	4	5	1	6
	label 2	0	45	5	2	3
	label 3	7	7	55	3	7
	label 4	2	3	5	43	5
	label 5	4	2	7	8	51

The confusion matrix was created using the validation dataset after the Neural Network model had been trained.

5.4 Classification Report

The classification report [81], using the Scikit-learn library [73], provides information on four categories, which are precision, recall, f1-score and support. Information from the confusion matrix is used to calculate precision and recall.

Precision is how effective the classifier is at not falsely labelling a positive outcome. Equation 5.1 [73, 81] is used to calculate this:

$$precision = \frac{\text{no. of true positive cases}}{\text{no. of all the positive cases}} \quad \text{(Eq 5.1)}$$

Where, $\text{no. of all the positive cases} = \text{true positive} + \text{false positive}$

The recall is the ability of a classifier to find all the positive instances. This includes both true-positive and false-negative outcomes. Equation 5.2 shows how the recall values are calculated [73, 81].

$$\text{Recall} = \frac{\text{no. of true positive cases}}{\text{no. } of\ true\ positives + \text{no. } of\ false\ negatives} \quad \text{(Eq 5.2)}$$

F1-score is a weighted harmonic mean value using both Precision and Recall values. This measurement is normally used when the dataset has an imbalanced distribution of data between the labels. Equation 5.3 shows how f1-score values are calculated. Finally, support is the number of figures of each class label that were assigned to the evaluation dataset [73, 81].

$$\text{f1 score} = \frac{(\text{Precision} * \text{Recall}) * 2}{\text{Precision} + \text{Recall}} \quad \text{(Eq 5.3)}$$

The classification report matrix was created using the validation dataset after the Neural Network model had been trained, and after the confusion matrix was created, since it requires data from it [73, 81].

5.5 Proposed Models

Using the proposed algorithm, I investigated different models on the two datasets discussed in section 5.2. Both models share a similar architecture. Five convolutional layers are used, with both using ReLU activation functions to link them together. These are followed by a flatten layer and a dense layer which connect to the output layer. The dense layer also uses a ReLU activation function, while the output layer used softmax. Both models also used an Adam optimizer. However, there were differences, including the learning rate and the size of the dense layer. Investigations found that each model suffered when using the parameter values from the other model. Learning rates $1x10^{-3}$ and $1x10^{-4}$ were used on the driver distraction and fatigue models, respectively. While the number of neurons in the dense layer was 2048 for the fatigue model and 1024 for the driver distraction model. With these variations in the architecture, the number of trainable parameters differed. Finally, The model for driver distraction had 1,071,362 trainable parameters, while the model trained on the fatigue dataset had 2,024,698 trainable parameters.

The driver distraction model achieved 97.48% training accuracy (loss of 0.0850) and a validation accuracy of 96.59% (loss of 0.1210). Finally, during evaluation, an accuracy of 99.58% accuracy, with 0.015 loss, was achieved. Figure 5.1 and table 5.2 shows the confusion matrix and classification report for the driver distraction model, respectively. The precision for the ten labels averaged at 98%, with the lowest being 96% and the highest at 100%. The high accuracy in predicting the correct label explains the recall and the f1-score reaching 98% average.

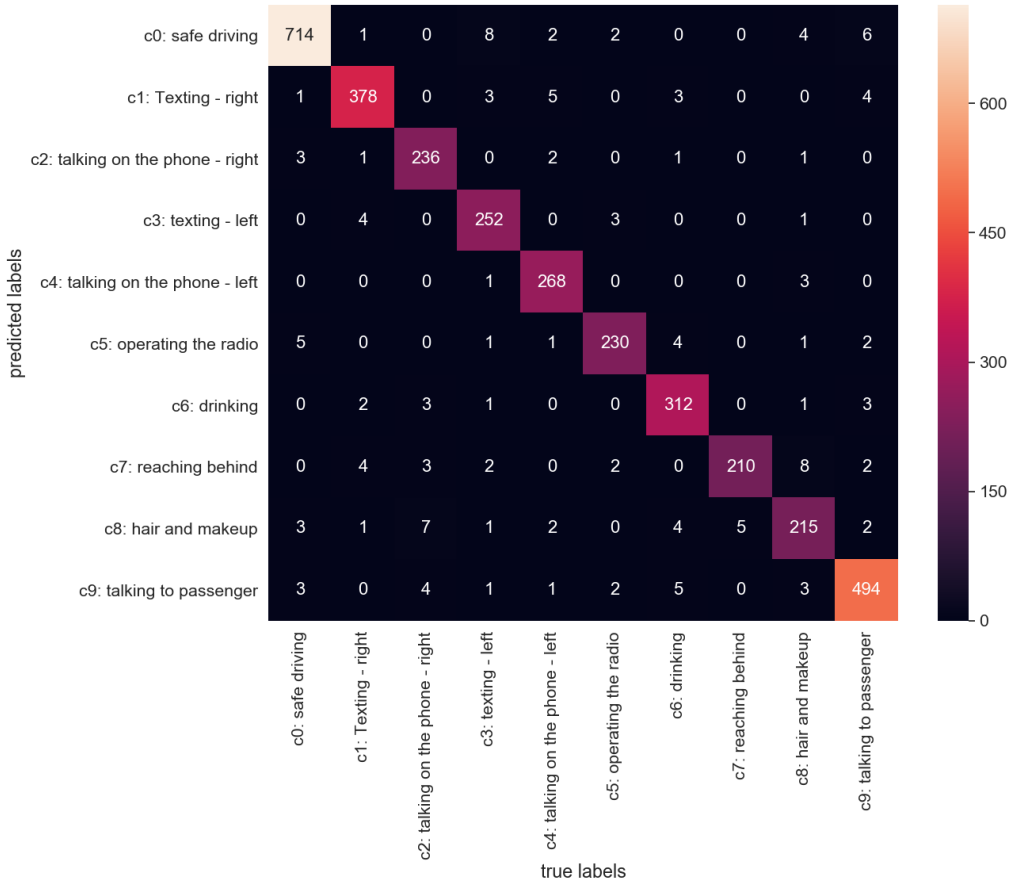

Fig. 5.1 Confusion Matrix heat map for the driver distraction model.

Table. 5.2 Classification Report for the driver distraction model.

labels	precision	recall	f1-score	support
c0	0.97	0.96	0.96	737
c1	0.99	0.99	0.99	394
c2	0.97	0.99	0.98	244
c3	0.97	1	0.99	260
c4	0.99	0.99	0.99	272
c5	0.99	0.98	0.99	244
c6	0.99	0.98	0.99	322
c7	1	0.97	0.98	231
c8	0.94	0.95	0.95	240
c9	0.96	0.98	0.97	514
average/total	0.98	0.98	0.98	3458

For the fatigue detection model, training reached 100% accuracy with 3.184 $x10^{-4}$ loss. Validation achieved 69.31% accuracy with 2.828 loss. Finally, evaluation produced 69.17% accuracy with a loss of 4.95. Figure 5.2 and table 5.3 show the confusion matrix and classification report for the fatigue model, respectively. The heat map of the confusion matrix shows high accuracy for correctly labelling drowsy images. However, the precision score for this label reached 66%, while the alert label received 79% accuracy. The reason for this is that the model incorrectly predicted 184 images as drowsy when they were actually images labelled alert. Of the 398 drowsy images, the model predicted 360 correctly, and only 38 images were incorrectly identified as alert. Hence this label received a high recall of 90% (equation 5.2). The alert label only managed 44% because, while it accurately identified 144 images as alert, 184 were identified incorrectly as drowsy as well. The high amount of incorrectly labelling drowsy images caused the alert label to have a lower recall value. Thereby impacting the f1-score, which produced weighted harmonic mean values for both labels using the precision and recall calculations. Since each label excelled in only one of either precision or recall, the f1-score averaged the drowsy label higher than its precision; and it averaged the alert label higher than its recall. Had the model not over labelled images as drowsy, the overall f1-score would have improved both label predictions, with the alert label improving the most. The model overfitting can explain the over labelling during training. Looking at the classification report and confusion matrix, the model picked up too many (unnecessary) features (overfitting) when training on both sets of labels and ended up over-identifying drowsy images as a result [60].

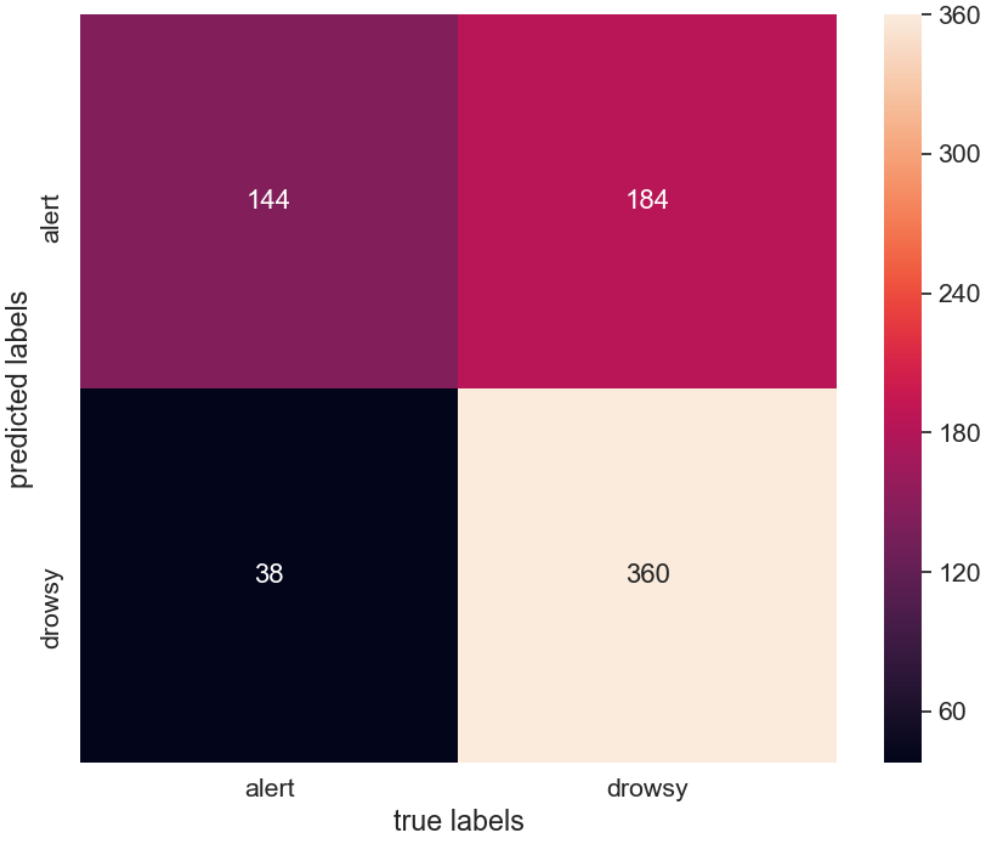

Fig. 5.2 Confusion Matrix heat map for the fatigue model.

Table. 5.3 Classification Report for the fatigue model.

labels	precision	recall	f1-score	support
alert	0.79	0.44	0.56	328
drowsy	0.66	0.9	0.76	398
average / total	0.72	0.69	0.67	726

5.6 Parameters

Both datasets, discussed in section 5.2, were used to determine the parameters for the proposed model that would give the highest training and validation accuracy for each dataset. Along with investigating variable values, different optimizers and activation functions were looked at to see how they would affect the accuracy and loss for each model.

The number of first layer filters (num_first_layer_filters) is the number of filters the convolutional layer will learn. As discussed in chapter four, this variable determined the number of outputs and the number of trainable parameters for each convolutional layer.

5.6.1 Number of neurons in the Dense Layer

The number of dense layer neurons is the number of neurons in each dense layer [58]. For both proposed Neural Networks the there was one dense layer, connected between the output layer and the last convolutional layer. To observe how the number of neurons in the dense layer affected the accuracy and loss, three models were trained and validated with appropriately different parameter values. Figure 5.1 shows the list of parameters for the model trained with 2048 neurones in the dense layer. For each model, all of the parameters were kept the same, except for the parameter num_dense_layer_neurones, which assigned 512, 1024 and 2048. Hence it was the independent variable for this investigation.

```
epoch = 5
learning_rate = 0.001
num_first_layer_filters=128
kernel_size=3
num_dense_layer_neurons=2048
no_labels = 10
```

Fig. 5.3 Parameter values for the Neural Network assigned 2048 neurons.

Figures 5.2 and 5.3 show the training accuracy and training loss, for the models with 512, 1024 and 2048 neurons in the dense layer, reached 94.56% with 0.1734 loss, 95.20% with 0.1631 loss and 95.48% with 0.1445 respectively. The results

determine that, as the number of neurons in the dense layer increased, training accuracy also increased (figure 5.2). With this increase in training accuracy, the loss reduced, almost minimising to zero (figure 5.3).

During Validation, the accuracy reached 94.460%, 94.04% and 95.87% for dense neuron numbers of 512, 1024 and 2048, respectively (figure 5.4). The validation loss was minimised to 0.1778, 0.2014, 0.1385, and for dense neuron numbers 512, 1024 and 2048, respectively (figure 5.5).

The only situation where the validation showed better accuracy and minimised the loss more than during training, was when the dense layer neurons were set to 2048. There was an increase in accuracy by 0.39% and the loss was minimised by 0.006 more. Furthermore, the validation results showed that the 512 neurons showed improved results compare to 1024 neurons. They produced an accuracy increase of 0.42% and an improved minimisation of the loss by 0.0236.

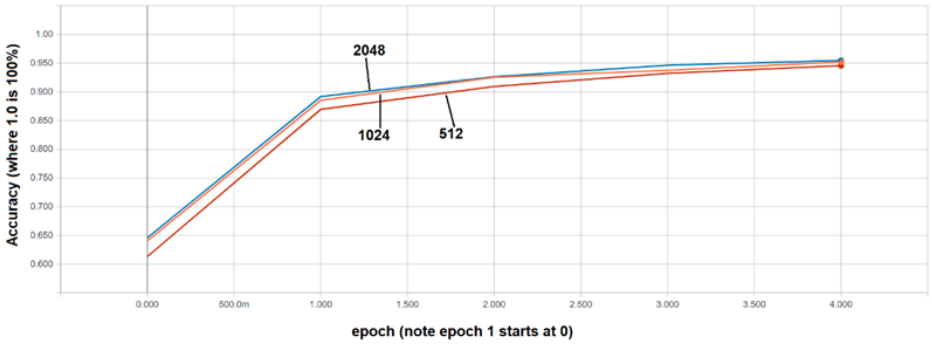

Fig. 5.4 Training Accuracy comparison of the number of neurons in the dense layer.

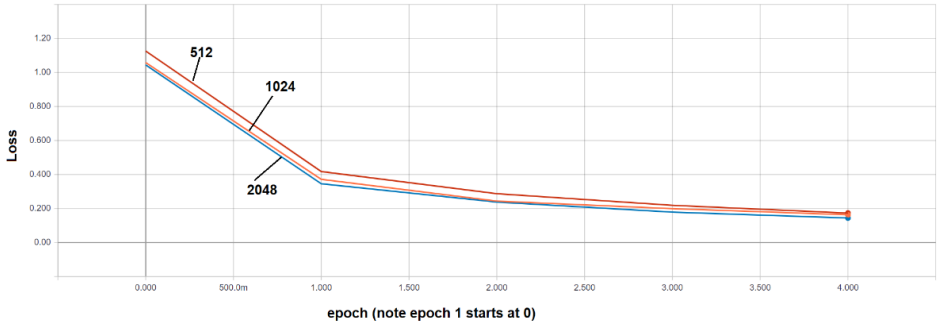

Fig. 5.5 Training Loss comparison of the number of neurons in the dense layer.

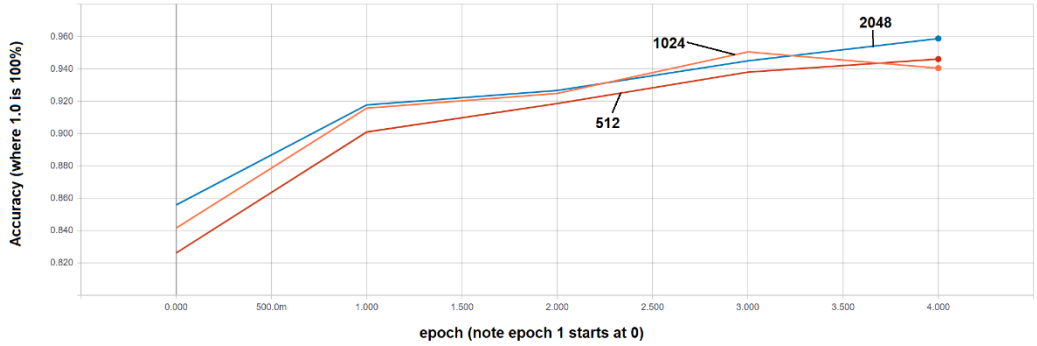

Fig. 5.6 Validation accuracy comparison of the number of neurons in the dense layer.

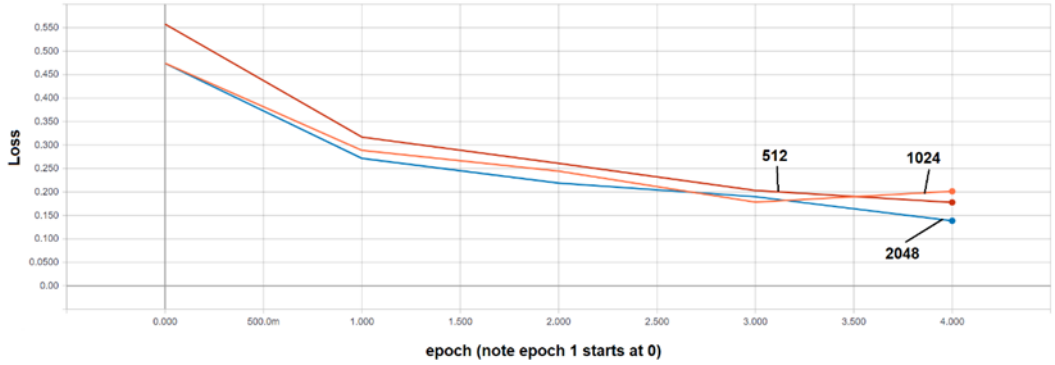

Fig. 5.7 Validation Loss comparison of the number of neurons in the dense layer.

The results indicate that the higher the number of neurons, the better accuracy and minimisation of the loss will be. Since there is an improvement the model with 512 neurons had over the model with 1024, indicates there is justification to suggests the relationship is not entirely linear, and there may be conditions where too many

neurons in the dense layer can negatively impact the accuracy. However, it is crucial to consider that validation can be considered a prediction and that because the differences between the two models (512 and 1024) are small, if validation was repeated, the outcome may reverse. Looking at the results collectively, the overall improvement in accuracy and loss minimisation is minimal. Hence, it may not justify the increase in computational power and time needed by doubling or quadrupling the size of the dense layer. Lastly, an important observation is that the training accuracy and its respective validation accuracy do not differ much from each other. Hence it is likely the model was successful in fitting and that there was minimal overfitting of the data by each model.

When training the fatigue dataset (results in appendix 8.2.1), the accuracy during validation was highest with 2048 neurons in the dense layer. However, values higher than this (3072 and 4096) showed the validation accuracies decreasing. This suggests there is an upper limit to how large the dense layer can be before it starts to affect the validation accuracies negatively. Also, the validation loss increased with validation accuracy as more epochs were run. This occurs when the model becomes less sure of the predictions when it tries to label an input image correctly. Hence the model has started to overfit on the training dataset, becoming too confident during training and struggles with validation. Table 5.4 summarises the results for each model with a different number of neurons, with the best results for each accuracy and loss highlighted.

Table. 5.4 Accuracy and loss of different dense layer sizes for driver distraction.

No. neurons in dense layer for each model	training accuracy	training loss	validation accuracy	validation loss
512	0.9456	0.1734	0.9460	0.1778
1024	0.9520	0.1631	0.9404	0.2014
2048	0.9548	0.1445	0.9587	0.1385

5.6.2 Learning Rate

The learning rate is a value (between 0 and 1) assigned to Neural Network models. This value determines how much the weights are updated and applied to the layers in the Neural Network model during training. The learning rate controls how quickly the model adapts to the problems it faces, in the proposed algorithm, the Neural

Network model was attempting to identify the action occurring in the image or the fatigue state of the participant, depending on the dataset the model was training on at the time.

Typically the lower the learning rate, the more epochs needed since the change in weights is less significant [56, 68]. The larger the learning rate, the more significant the change in weights between epochs, and hence fewer epochs are needed. Figure 5.8 shows the input values used while investigating the learning rate.

```
epoch = 5
batch_size = 64
learning_rate = 0.004
num_first_layer_filters=128
kernel_size=3
num_dense_layer_neurons=1024
no_labels = 10
```

Fig. 5.8 Input values for the Neural Network.

Figures 5.9, 5.10, 5.11 and 5.12 show the accuracy and loss for both training and validation (for the learning rate models). For these sets of results, the learning rates were set to $1x10^{-2}$,$1x10^{-3}$,$1x10^{-4}$,$1x10^{-5}$. Both learning rates, $1x10^{-3}$ and $1x10^{-4}$, followed similar trends, with the former showing most improvements to the accuracy and loss minimization, during both training and validation. Furthermore, these models showed that their respective validation values and training accuracies to be similar, suggesting the models fit the training data well. The learning rate at $1x10^{-5}$ had less accuracy during both training and validation. Looking closely at figure 5.9, the trend for $1x10^{-5}$ is going upwards as the last epoch is completed. If more epochs were run, it can be suggested that the training accuracy and validation accuracy could reach higher accuracies, similar to learning rates $1x10^{-3}$ and $1x10^{-4}$. However, when trying to keep computational time as low as possible, it is preferable to use larger learning rates to lower the number of epochs required to reach high accuracy, in both training and validation.

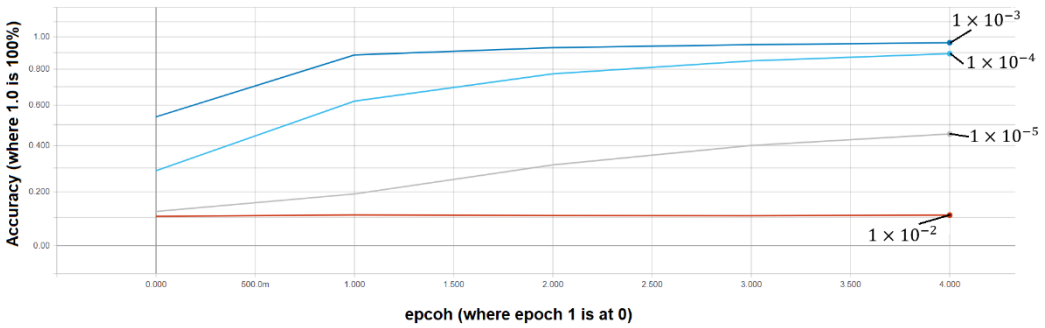

Fig. 5.9 Training accuracy comparison for differing learning rates

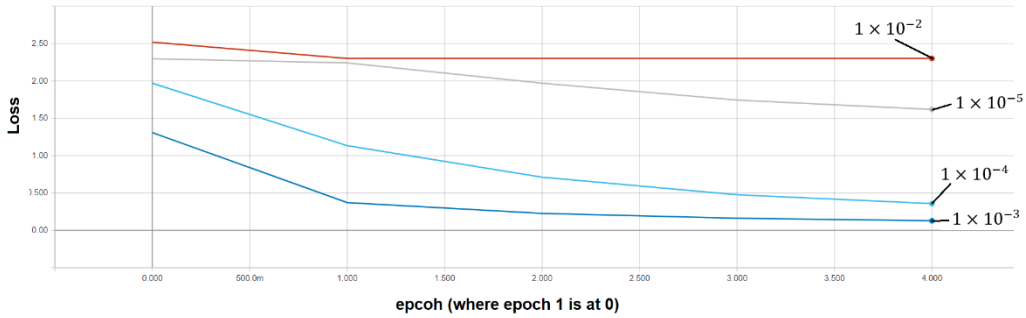

Fig. 5.10 Training loss comparison for differing learning rates

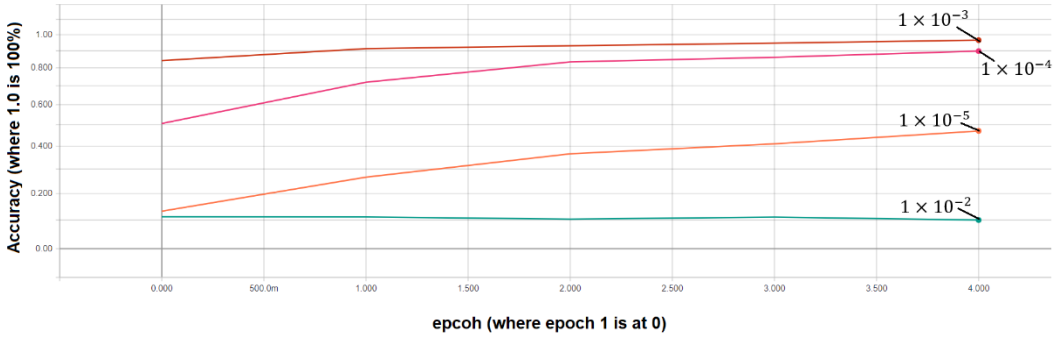

Fig. 5.11 Validation accuracy comparison for differing learning rates.

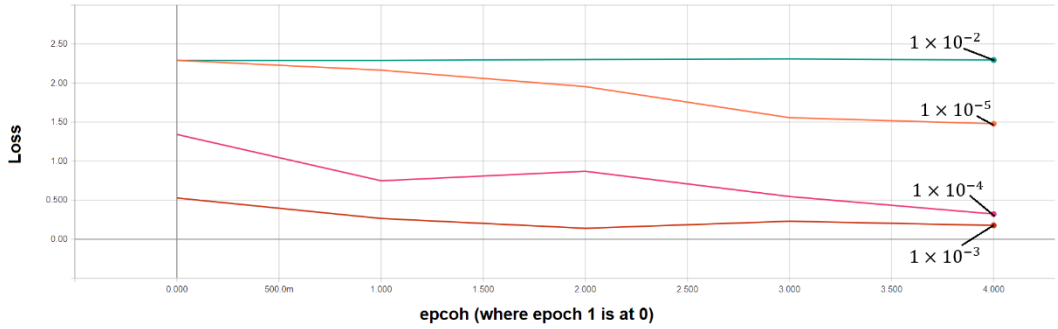

Fig. 5.12 Validation loss comparison for differing learning rates

The model, given a learning rate of $1x10^{-2}$, showed an inability to accurately train, showing just over 10% accuracy and thus, unable to minimise the loss much. This suggests that the model did not have enough trainable parameters and resulted in underfitting. Lastly, the results clearly show how important selecting the correct learning rate is for CNN models. Table 5.5 summarises these results.

When training on the fatigue dataset, the learning rates followed a similar pattern, except that the highest accuracy was produced using $1x10^{-4}$ as the learning rate. Smaller learning rates (appendix 8.2.2) than this showed reduced accuracies, although they were on an upward trajectory. Hence while a lower learning rate can be used, to save computation time, the more optimum choice was at $1x10^{-4}$. With both datasets requiring different learning rates to reach optimum accuracies, an adaptive approach would be more efficient. This is discussed in chapter six.

Table. 5.5 Summary of the accuracy and loss of different learning rates.

learning rate models	training accuracy	training loss	validation accuracy	validation loss
$1x10^{-2}$	0.1092	2.301	0.0996	2.295
$1x10^{-3}$	0.9626	0.1284	0.9656	0.1773
$1x10^{-4}$	0.8933	0.3578	0.8984	0.3225
$1x10^{-5}$	0.4553	1.618	0.4705	1.479

5.6.3 Number of epochs

One epoch is when the entire dataset is passed forward and backwards through the Neural Network once. Typically there are hundreds of thousands of data points, which is too much for the computer to pass them all in one go. Instead, the Neural Network re-arranges the data into small groups known as batches [56, 57]. The parameter the proposed algorithm used to assign the size of the batch was called bacth_size. For all the investigations made in this thesis, the batch size was set at 64, as shown in figure 5.13. The only effect the batch size had on the training and testing of the model was to illustrate how many iterations were needed to pass through all the images for each epoch. Hence it did not affect the accuracy. While investigating driver distraction, the epoch was increased to 10 from 5, as also shown in figure 5.13.

Figure 5.14 shows the training accuracy, for the 10-epoch model, increased to 97.48%, an increase of 2.28% compared to the 5-epoch model. The increase in accuracy is expected with more epochs, since the weights will be updated more often, and the training is repeated more often. The training and validation losses (figures 4.15 and 5.17) were minimised as expected, with the epoch10 model outperforming the 5-epoch model. Finally, with the validation accuracies (figures 4.15 and 4.16), for both models, was very close to the training accuracies, the models fitted to the training data well enough for similar accuracies when tested during validation. Table 5.6 summarises these results.

```
epoch = 10
batch_size = 64
learning_rate = 0.002
num_first_layer_filters=128
kernel_size=3
num_dense_layer_neurons=1024
no_labels = 10
```

Fig. 5.13 Input values for the Neural Network.

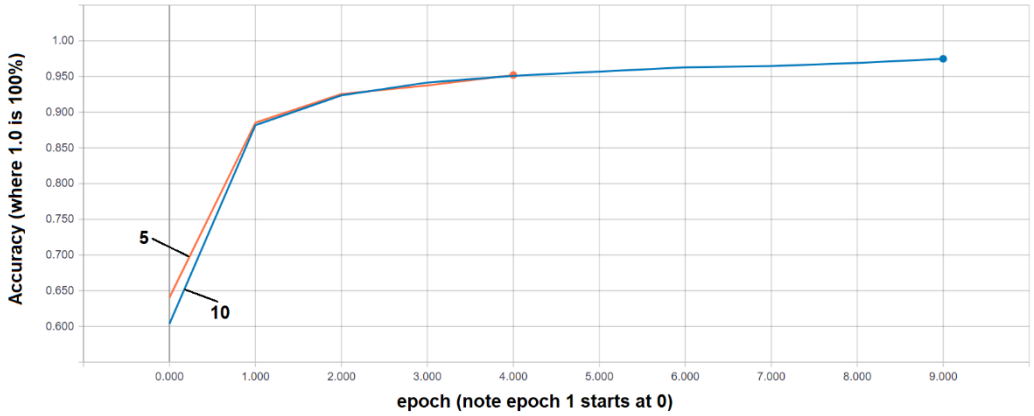

Fig. 5.14 Training accuracy for different epoch values.

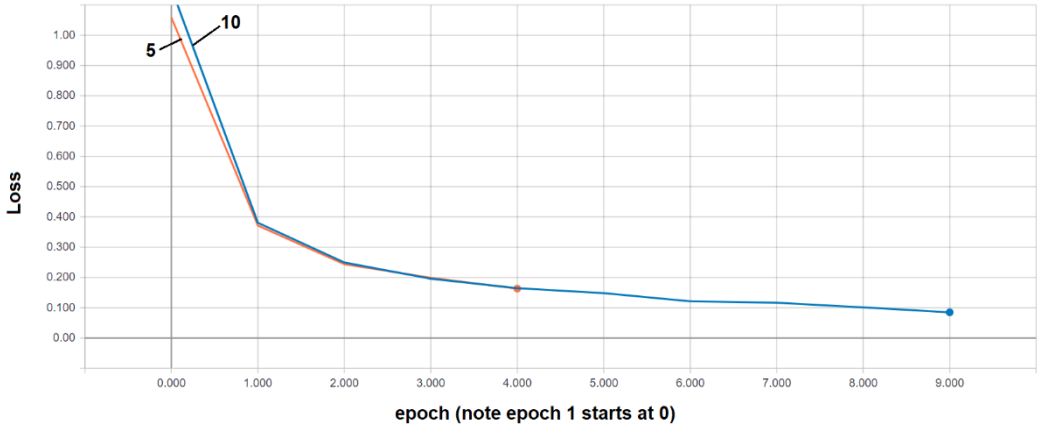

Fig. 5.15 Training loss for different epoch values.

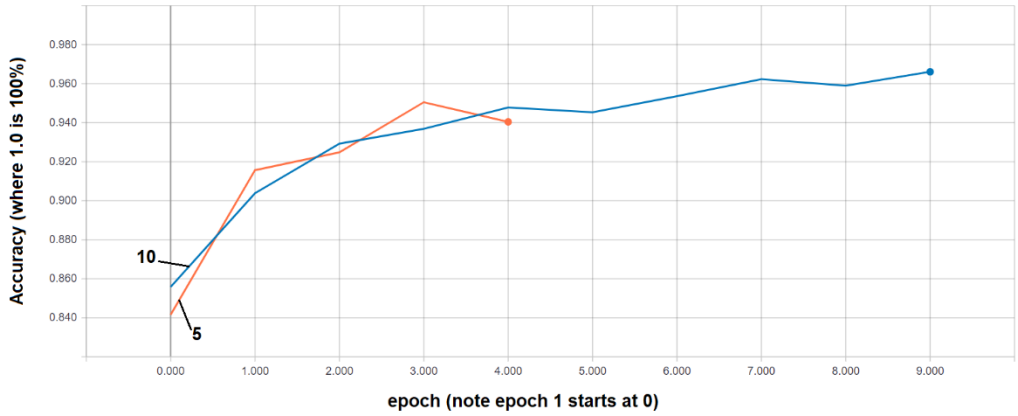

Fig. 5.16 Validation accuracy for different epoch values.

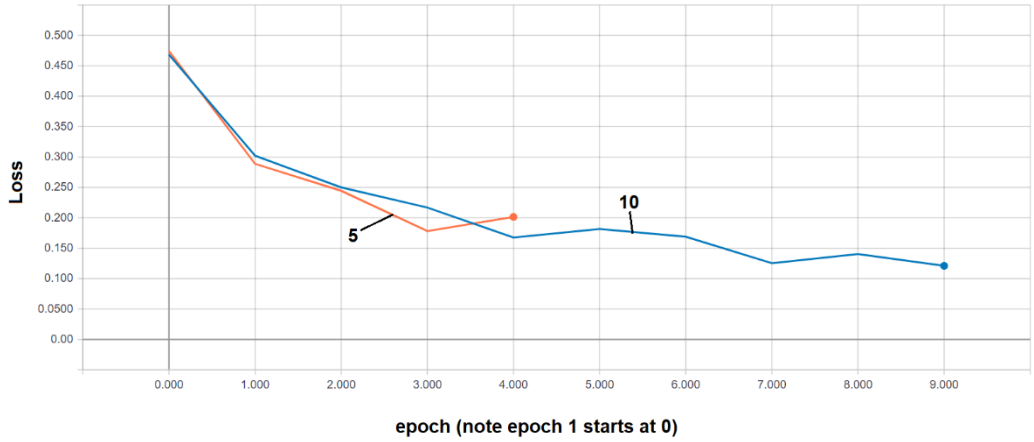

Fig. 5.17 Validation loss for different epoch values.

Table. 5.6 Summary of the accuracy and loss for different epochs

epoch assigned for each model	training accuracy	training loss	validation accuracy	validation loss
5	0.9520	0.1631	0.9404	0.2014
10	0.9748	0.0850	0.9661	0.1210

5.6.4 Kernel size

A kernel is a matrix of weights that go through the image matrix, applying them to the pixel values on the image to produce an output matrix [35, 76, 78]. The real kernel size is a two-dimensional vector of the same given value. For instance, in figure 5.1, the kernel size is 3x3.

The investigation involved observing changes to the kernel size had on the accuracy during training and validation. Figure 5.18 shows the parameters used, with the kernel size changing for each model.

```
epoch = 5
batch_size = 64
learning_rate = 0.002
kernel_size=5
num_dense_layer_neurons=1024
no_labels = 10
```

Fig. 5.18 Input values for the Neural Network.

Figures 5.19, 5.20, 5.21 and 5.22 showed the accuracies and loss minimisations for both training and validation. The graphs show a comparison between 2 models with kernel sizes 5x5 and 3x3. The 5x5 kernel model showed a training accuracy of 11.1%. The low training accuracy suggested the model failed to fit the data (underfitting), leading to low validation accuracy and failed to minimise the loss during the entire simulation (figures 5.20 and 5.22). In contrast, the 3x3 showed high accuracy, but more importantly, an ability for the model to learn, unlike the 5x5 kernel model.

Selecting the correct kernel size is key to a functioning convolutional Neural Network [35, 76, 78]. Typically, the smallest possible kernel is selected (3x3) for high accuracy, shown in figures 5.19 and 5.21. As the kernel size increase, small details in the image can get overlooked, resulting in less accurate predictions. Also, there may be multiple features in the image, and a larger kernel risks not discovering the feature in other areas of the image. Lastly, while choosing an even-sized kernel is possible, odd-sized kernels are preferred. When the kernel has a central element or pixel(i.e. an odd-sized kernel), the rest of the pixels in the kernel are symmetrically around the central pixel. This makes it easier to slide the kernel across the whole image by one pixel at a time. An even kernel has to account for distortion across the image, making the process unnecessarily complicated [35, 76, 78].

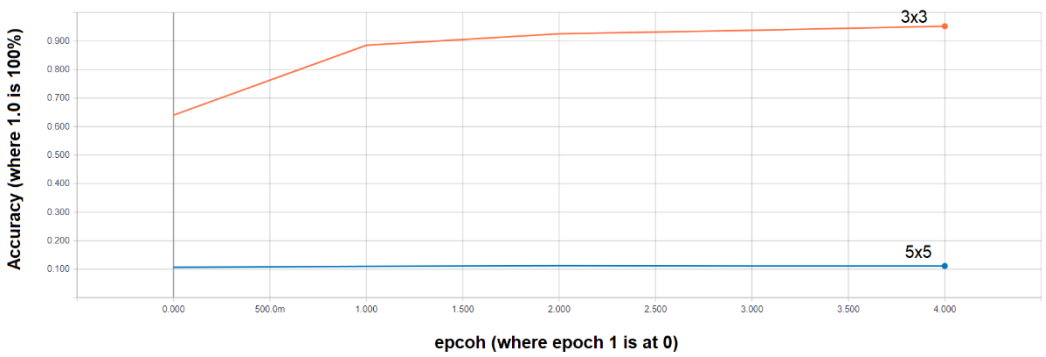

Fig. 5.19 Training accuracy for different kernel sizes.

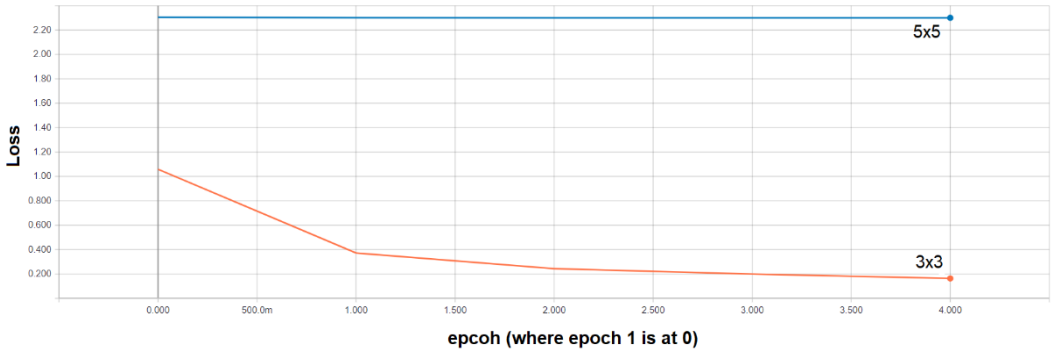

Fig. 5.20 Training loss for different kernel sizes.

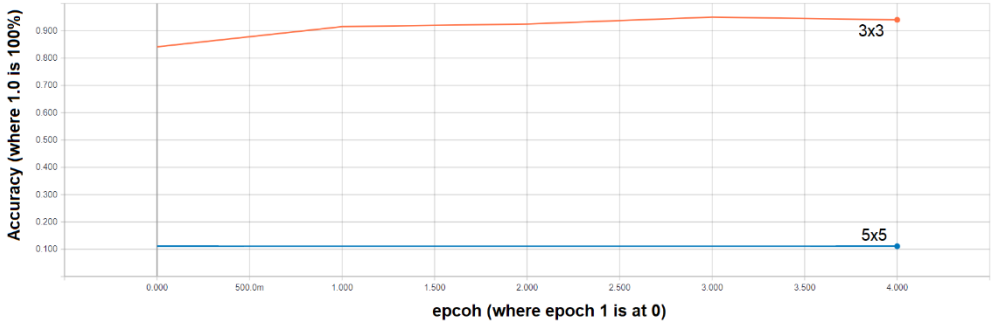

Fig. 5.21 Validation accuracy for different kernel sizes.

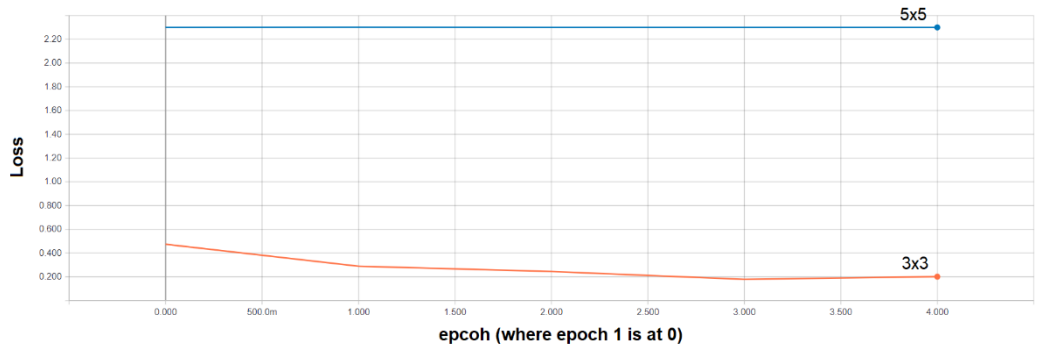

Fig. 5.22 Validation loss for different kernel sizes.

5.6.5 Number of labels

The no_labels is the number of labels the Neural Network is being trained on. Regarding driver distraction, The labels were Safe Driving, Texting Right, Talking on the Phone - Right, Texting Left, Talking on the Phone - Left, Operating The Radio, Drinking, Reaching Behind, Hair and Makeup and Talking to Passenger. For fatigue, the labels were alert, low vigilant and drowsy.

The number of labels a model trains on can affect the accuracy (training and validation). This is due to the model requiring more trainable parameters, and as such, increases the likeliness of overfitting. Since the driver distraction database had ten labels, I decided to train each label separately using the same model. Figures 5.24, 5.25, 5.26 and 5.27 showed the accuracy and loss for both the training and validation. Label c6 (drinking) produced the highest training and validation accuracy at 98.88% and 98.73, respectively. In comparison, c2 (talking on the phone - right) produced the lowest training and validation accuracies at 89.74% and 90.08% respectively. Table 5.7 summarises the results for each label.

Note each label was trained against all the labels, which were combined into one label. As a result, there were two labels that the Neural Network was trained on each time. One label was what the model was training to identify (call this the YES label) and the second label contained examples of what it should consider is the incorrect label (call this the NO label). For example, label c5 (operating the radio) was trained against one other label containing the images of c1 to c9. This meant the model, during training, was learning to differentiate between operating the radio

and everything else. The result of this was that the NO label had much more data samples in it than the YES label. With ten models trained to identify one specific label, testing each image was carried out on each model with the one providing the highest predictability assigned the label. This method of training is typically carried out when the dataset is not uniform among each label, in terms of the number of samples per label.

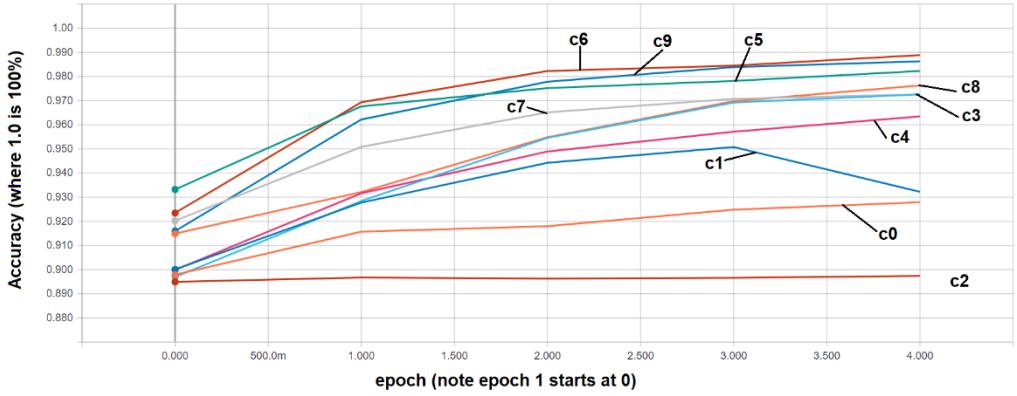

Fig. 5.23 Training accuracy of the separate labels.

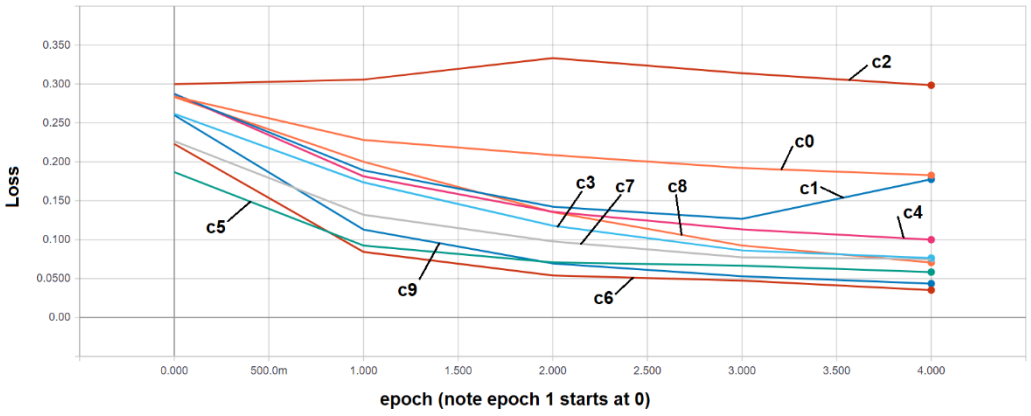

Fig. 5.24 Training Loss of the separate labels.

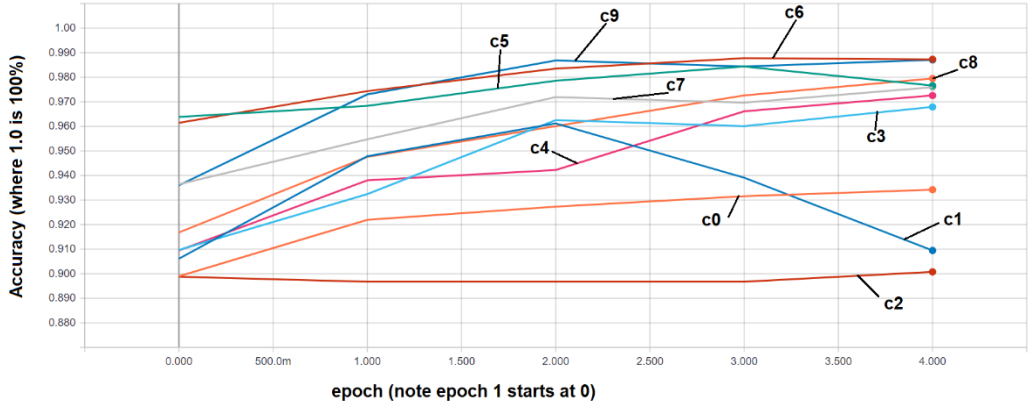

Fig. 5.25 Validation accuracy of the separate labels.

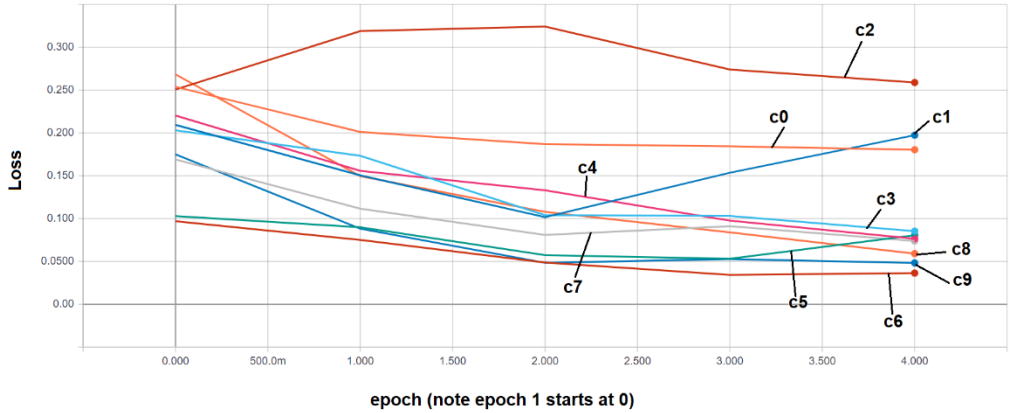

Fig. 5.26 Validation loss of the separate labels.

Table. 5.7 Results after training and validating each folder, one at time.

One label trained at a time	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
c0: safe driving	0.9279	0.1827	0.9342	0.1805
c1: texting - right	0.9323	0.1775	0.9095	0.1974
c2: talking on the phone - right	0.8974	0.2984	0.9008	0.2589
c3: texting - left	0.9726	0.0764	0.9679	0.0855
c4: talking on the phone - left	0.9635	0.0999	0.9726	0.0765
c5: operating the radio	0.9823	0.0584	0.9766	0.0807
c6: drinking	0.9888	0.0352	0.9873	0.0365
c7: reaching behind	0.9729	0.0750	0.9759	0.0739
c8: hair and makeup	0.9763	0.0706	0.9795	0.0592
c9: talking to passenger	0.9863	0.0435	0.9871	0.0483

5.6.6 Optimizers

Various optimizers (discussed in chapter three) were investigated to observe their effect on accuracy and loss, along with computational time. Figure 5.27 showed the training accuracy for the optimizers when the model was training on the driver distraction dataset [10, 11]. All the optimizers performed similarly to each other (in terms of training accuracy, with exception to the SGD optimizer. Typically, SGD optimizers work better when small learning rates are used [62]. This suggests that a learning rate of 0.001 was not low enough for SGD to operate efficiently. Looking closer at the rest of the optimizers, Nadam and Adadelta produced slightly higher accuracies, compared to Adam [64-66]. However, with respect computation time, Adam was the quickest to finish training. The loss minimisation of each of the optimizers followed expectations (figure 5.28), with SGD minimising the loss the least and the rest performing similarly to one another. Figures 5.29 and 5.30 show, accuracy and loss during validation, respectively. Here the Adam optimizer produced a higher validation accuracy and minimised the validation loss more than the rest of the optimizers. Except for SGD, the differences in accuracy and loss were small, and potentially any could be used to train on the distraction dataset. However computationally, Adam was faster and hence was used for the proposed model.

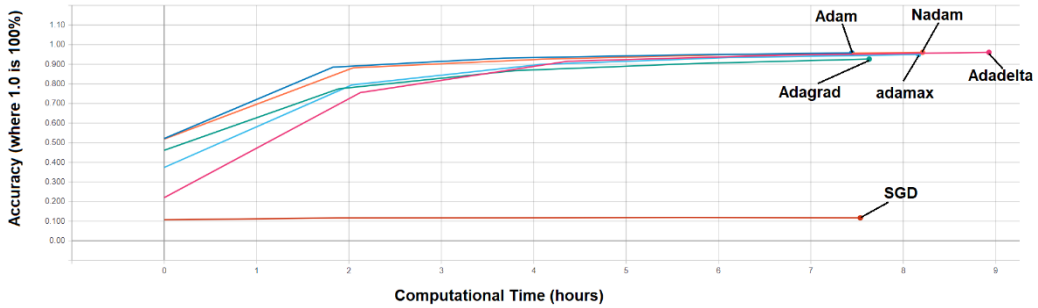

Fig. 5.27 Training accuracy for different optimizers for driver distraction dataset.

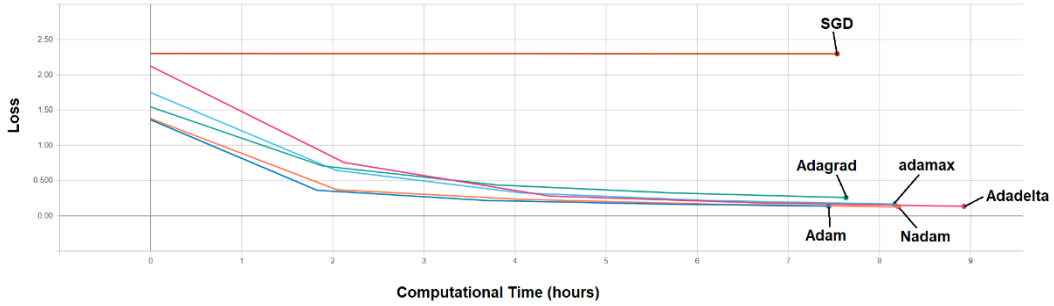

Fig. 5.28 Training Loss for different optimizers for driver distraction dataset.

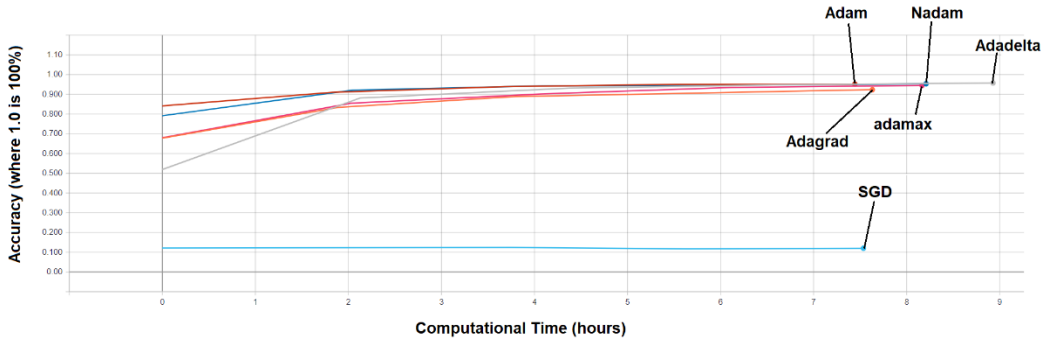

Fig. 5.29 Validation accuracy for different optimizers for driver distraction dataset.

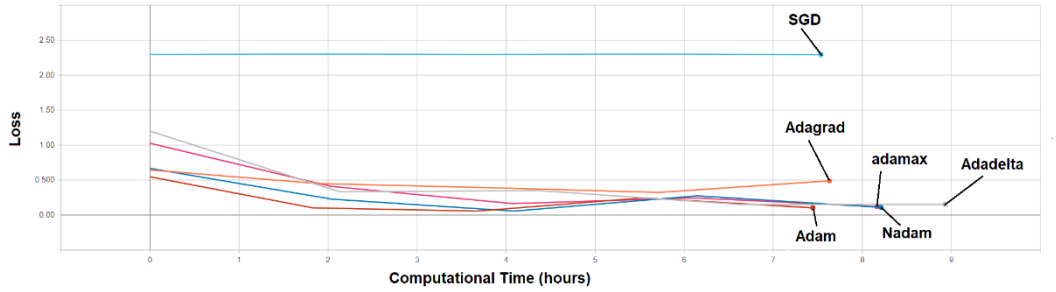

Fig. 5.30 Validation Loss for different optimizers for driver distraction dataset.

When training on the fatigue dataset [9] the optimizers, Adam, Nadam and Adamax performed the best while SGD, Adadelta and Adagrad only managed to achieve about 55% accuracy (appendix 8.2.3). The performance of the SGD optimizer differed between the two datasets was with the SGD optimizer. While SGD failed to train when detecting driver distraction, it managed 55% accuracy when detecting

fatigue. The difference affecting the optimizers in both datasets was the learning rate. For the model detecting fatigue, the learning rate was lower (10^{th} smaller) compared to driver distraction. Since the learning rate determines the steps the optimizer takes when trying to find the minimum (chapter three), then it suggests that while training for driver distraction, the larger learning rate was too big for the model to learn how to distinguish between the labels. Looking at validation accuracy and loss for fatigue detection, the accuracy either fluctuated by up to nearly 20% or changed very little. Finally, while Nadam, maintained a higher accuracy for most of the computational time, Adam produced a very similar accuracy after the 1^{st} epoch, and never improved on it. This suggests that the models testing different optimizers (appendix 8.2.3) over fitted, hence relied on training too much and struggled with validation. However, because of the comparable accuracies between adam and Nadam, Adam was decided for the proposed model, due to its faster computational time.

5.6.7 Activation Functions

Activation functions are used to scale down the output of each neuron. Typically, two activation functions are used in Convolutional Neural Networks [52, 82]. One is used between the convolutional layers and dense layer, while the second is used at the output layer. Studies show that ReLU is commonly used for activating convolutional layers, which some of my investigations also determined [46, 50, 82, 83]. The output activation function is different depending on the number of labels used. When more than two labels are in the dataset, softmax is recommended [46, 50, 82], as it provides a probability for each label likely being correct, and the one with the highest probability is used to predict the label. Sigmoid is typically used when there are two labels. This is due to this activation function having its output trend to the extremes (either very close to zero or one) [46, 50, 82]. Figure 5.31 shows different activation functions being used at the output. With these results, only two labels were used when training on the fatigue dataset (aware and fatigued).

What is clear is that sigmoid and softmax functioned the best during training, both in accuracy and loss (figures 5.31 and 5.32). While ReLU and tanh failed to perform as well. What was interesting is that the sigmoid function, which is prefeed when

two labels are in the dataset, performed as well as the softmax function during training. However, during validation (figures 5.33 and 5.34), softmax produced a higher accuracy of around 69%, while sigmoid failed to produce any validation accuracy above 60%. While sigmoid required less computational time, the almost 10% difference in the accuracy, it was decided to use softmax as the output activation function.

I arrived at the same conclusion when investing these models for driver distraction. The model when ReLU was activating the CNNs along with a softmax output. The only issue that was discovered was that, for fatigue detection, validation loss increased for all the functions. This was due to the models suffering from overfitting, even though the loss never went above 1.

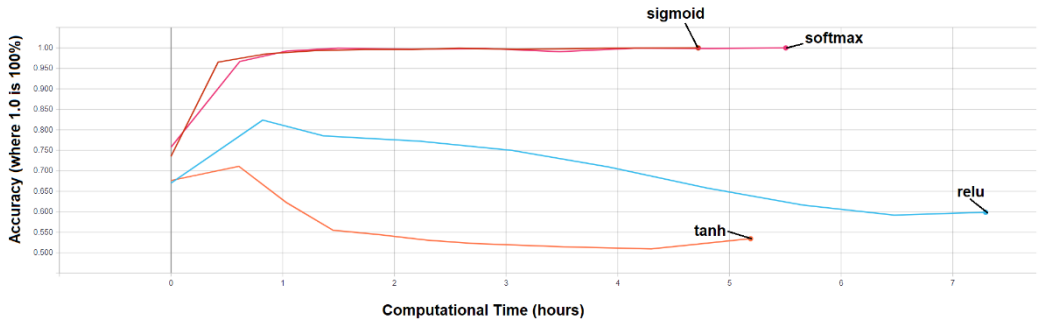

Fig. 5.31 Training Accuracy for different activation functions for fatigue dataset.

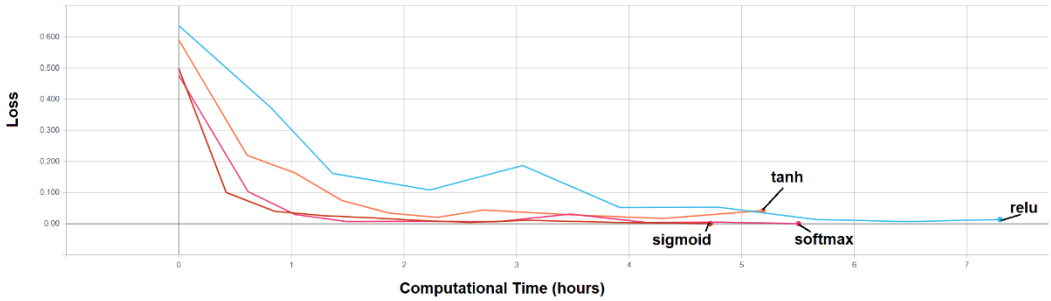

Fig. 5.32 Training Loss for different activation functions for fatigue dataset.

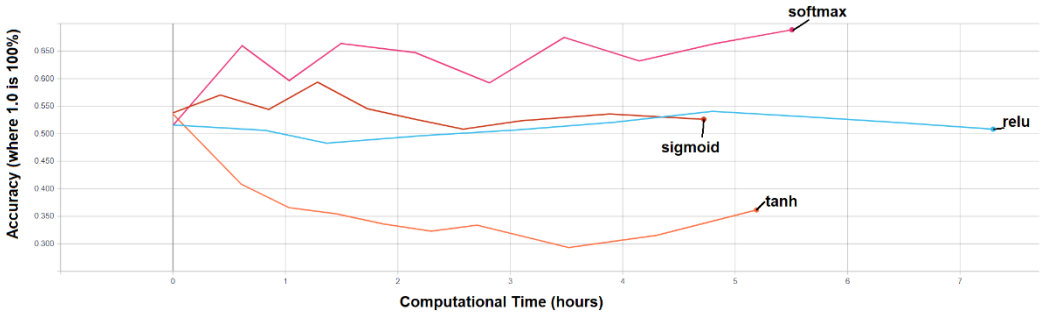

Fig. 5.33 Validation Accuracy for different activation functions for fatigue dataset.

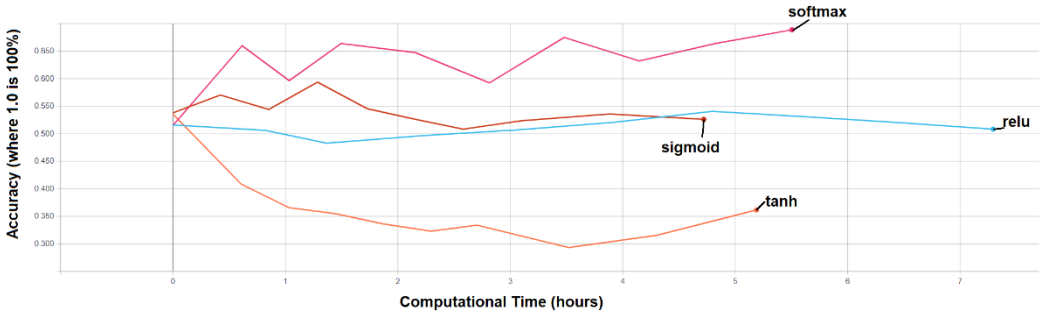

Fig. 5.34 Validation Loss for different activation functions for fatigue dataset.

5.7 Summary

Investigations of the parameters mentioned in this chapter were discussed regarding their impact during training and validation.

As the number of neurons in the dense layer increases, it improved the accuracy and loss minimisation during training and validation. However, there were suggestions in the validation results that indicated fewer neurons might yield better accuracy and loss minimisation, in some situations. This was true when investigating the same variable for fatigue detection. A too high neuron figure showed deterioration in accuracy for both training and validation. The similar conclusion found when looking at the learning rate. The smaller the learning rate showed an improvement in training and validation accuracy. The loss minimisation for both training and validation improved with lower learning rates as well. However, like the dense neuron layer, a too-small learning rate meant the model did not train fast enough and never reached the high accuracies in the computational time. With kernel size, the minimum size of 3x3 was recommended. Any higher and the models failed to train.

The more epochs the models were allowed to train, the higher the training and validation accuracies were. However, this comes at a cost to computational time. Considerations need to be taken to see if more epochs are necessary if the increase in accuracy is minimal. Similarly, the optimizers and activation function performance was based on computational time. For the optimizer, observations showed Adam performed faster and achieved similar results to the optimizers designed to improve on Adam, such as Nadam. The activation functions that were used included ReLU and softmax, where again they were computational faster.

The proposed model for training on the driver distraction dataset consisted of 5 convolutional layers with one dense layer (with activation function ReLU). The output layer used the activation function softmax, while the optimizer used was Adam. The resultant model meant that 1,071,362 trainable parameters were used. A validation accuracy of 96.59% was reached, along with the loss minimised to 0.1210. Evaluation saw the accuracy increase to 99.58% accuracy with 0.015 loss minimisation.

The proposed model for the fatigue dataset used consisted of 5 convolutional layers with one dense layer assigned 2048 neurons. The images were converted to grayscale during data generation. As a result, the model had 2,024,698 trainable parameters. Evaluation showed the model had 69.17% accuracy with 2.828 loss. The inability to minimise the loss more meant the validation accuracy could not reach the training accuracy of 100%.

With the two proposed models differing for both datasets suggest an adaptive algorithm is necessary to produce high accuracies when applied to multiple datasets. The proposed algorithm for this solution is discussed in the further work section of chapter six, conclusions.

5.8 References

[79] "State Farm Distracted Driver Detection." https://www.kaggle.com/c/state-farm-distracted-driver-detection/data (accessed August 2018).

[80] C. Sammut and G. I. Webb, *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[81] R. Garreta and G. Moncecchi, *Learning scikit-learn: machine learning in python*. Packt Publishing Ltd, 2013.

[82] P. R. Nicolas, *Scala for machine learning*. Packt Publishing Ltd, 2015.

[83] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375,* 2018.

Chapter 6: Conclusions and Future Work

6.1 Conclusion

This thesis discussed the merits of a Neural Network approach for solving the classification problem. The datasets used were designed on driver distraction and fatigue. The proposed algorithm could equally be used to classify different datasets. The algorithm also contains design considerations that allow the user to easily modify their Neural Network models to fit different datasets better. Along with the algorithm discussed in chapter four, this thesis proposed two original models that were trained on the driver distraction dataset from the American University in Cairo (AUC) [10, 11] and a fatigue dataset provided by The University of Texas at Arlington [9]. The model detecting driver distraction reached 99% accuracy during evaluation, while the model designed fatigue detection reached 71% evaluation accuracy.

Driving whilst fatigued contributes up to 20% of road accidents, according to ROSPA [1]. Research showed performance level, when fatigued, were affected similarly to individuals under the influence of alcohol. With travel safety always on the minds of the public, and advances in the vehicle industry, there is always a drive to improve transportation safety. The research this thesis has discussed could contribute to increasing the safety of transportation.

In chapter two, Conventional methods, such as Haar cascade classifiers and eigenfaces, were discussed as possible solutions to the classification problem. This research focused on detecting fatigue and distractive behaviour. While both conventional approaches have their advantages, the main disadvantage is that their methodology requires their respective algorithms to be designed specifically for the developer's intention. For example, a Haar cascade classifier designed to measure the blink rate, while very useful, would be restricted to its purpose. In 2018 I proposed a blink detection system [84], that used Haar cascade feature detection. For the detection itself, cadence software was used to implement a gate-level schematic. It was based on an asynchronous 6-bit based edge counter using D flipflops. The output of the system would trigger an alarm based on a blink rate threshold, which used a calculated average blink rate to compare with the current blink rate. This system proved effective at detecting and measuring blink rate; however, to detect driver distraction, it would need to be redesigned. The advantage with Neural Network solutions (discussed in chapter three) is they

provided a more efficient way to detect features. This is clear with the two proposed compiled models discussed in Chapter five. While both systems technically differ, their architecture is very similar, in that they both have the same number of convolutional layers and use one dense layer connected to the output. The trade-off with these systems is that they require large amounts of data to train on, which is proportional to computational time.

Chapter three discussed the principles of Neural Networks, looking specifically at deep machine learning using Convolutional Neural Networks (CNNs). The functionality of CNNs was discussed, along with the crucial elements of their models that would determine high accuracy. The critical areas of Neural Networks include the activation functions, Optimizers, Learning Rate, Convolutional Kernel, Epoch, forward and backward propagation. Activation functions are used to scale the output of each neuron to produce a probability at the output layer accurately. Typical CNNs use a Relu function for the convolutional layers and either a softmax or sigmoid function at the output player [52]. Softmax is used when multiple (more than 2) categories or labels are in the dataset. While sigmoid is typically used when two labels are present. Optimizers determine how effective the model is at determining the smallest gradient (gradient descent). This is known as loss minimisation. A vital aspect of this is the learning rate. Gradient descent is achieved by using a learning rate to determine how big the random steps the optimizer takes to achieve minimum loss. A small learning rate means the changes to the weights and biases for CNNs are less, while a larger learning rate means bigger jumps. Finally, a big factor that is considered in Neural Network models is the computational power. The more trainable parameters and the more epochs the model needs the more computational cost is required [35, 56, 75].

Chapter four discussed the proposed algorithm, presented in this thesis, which contains design considerations that allow the user to easily modify their Neural Network models to fit different datasets better. The algorithm relied on the Keras library and Tensorflow to produce the Neural Network models proposed in this thesis. It was developed using Python, version 3.6. The basic structure of the algorithm consists of first generating the training and validation datasets, compiling the model, training, validating and finally, evaluation and testing. During data generation, the training dataset can be augmented. This involves augmenting the input images, for example by zooming, shearing and horizontal flips. While these

sound-like legitimate changes that are feasible to occur, not all the available options were necessary. For example, vertical flip would produce images that are unlikely to be seen during testing; horizontal flip would make the subject of the image to appear on the opposite side. This would be commonly viewed when observing drivers in cars with the steering wheel on the left or the right.

Chapter five discussed the simulations that were run to obtain the most optimum models for each of the datasets trained the proposed algorithm. The proposed model for training on the driver distraction dataset, provided by AUC [10, 11], consisted of 5 convolutional layers with one dense layer (using activation function Relu) that connected the output layer to the last convolutional layer. The output layer used the activation function softmax, while the optimizer used was Adam [65, 66]. The resultant model had 1,071,362 trainable parameters. A training accuracy of 97.48% was reached, along with a validation accuracy of 96.59%. Loss minimisation for both training and validation was 0.0850 and 0.1210, respectively. Evaluation of this model showed an accuracy of 99.58%, with 0.015 loss. The proposed model for the fatigue dataset also consisted of 5 convolutional layers with one dense layer, although the dense layer was assigned 2048 neurons. As a result, the model had 2,024,698 trainable parameters. Evaluation of the model showed it had 69.17% accuracy with 4.95 loss.

Along with the proposed models, Observations and explanations on how some of the variables affected the accuracy and loss were also looked at in chapter five. This was important to determine the most optimum values for each proposed model. For the dense layer, the recommended value was different for both proposed models. The same was true for the learning rate. While the dense layer had to be bigger for fatigue detection, its learning rate had to be smaller (by a 10^{th}) with the use of the same optimizer, Adam [65, 66]. The fact that different learning rates were required further enhances the need for an adaptive approach. However, not all the parameters were unique to one of the proposed models. Along with the same optimizer, both shared the same kernel size and activation functions.

These conclusions lead to a final proposal which will improve on the algorithm and the models proposed in chapters four and five respectively. The approach requires a truly adaptive Neural Network, which I discuss further in the final section of this thesis.

6.2 Future Work

The proposed algorithm and models presented and discussed in this thesis were trained, validated and tested using images to identify driver distraction and fatigue. However, conventional technology (discussed in chapter two) use methods like Haar cascade [18, 19] and Eigenfaces [15] use for image processing in real-time. The ability for Neural Networks to be easily modified to train on different datasets gives them an advantage over Haar-cascade classifiers which lack flexibility.

The next step for this research is to develop an algorithm that can use videos as its inputs rather than images, known as video classification, available from the Keras library [50]. Regarding driver distraction, the videos would be analysed by extracting video frames from them that could recreate the motion, for example, using the radio. The Neural Network could then learn to identify the motion involved in using the radio, among other motions, so that during testing, it can identify the actions in real-time. This would open up the possibility of developing an on-board video processing system that can identify actions taken by the driver that is deemed distractive. This is similarly true for fatigue detection. The system could extract video frames that can be analysed to detect fatigue and then be used in real-time.

As discussed in chapter five, I propose an adaptive machine learning algorithm to improve the machine learning accuracies. My investigations on both the AUC dataset [10, 11] and the UTA-RLDD dataset [9] showed that Neural Network parameters are unique when building a model that produces a high accuracy rating during training and validation. I propose a single algorithm that, after each epoch updates various parameters, such as the learning rate and the number of neurons in the dense layer, is updated depending on how the training and validation accuracies change each time the training database is sent throughout the Neural Network model. Hence the system would involve parameter adaptation. As the model would train and validate after an epoch, the difference in accuracy to the previous epoch would cause the model to update at least one of its parameters. The change would be dependent on the outcome of the epoch. My research into the functionality of Neural Networks presented in chapters three and five have brought me to conclude that there are several parameters to consider.

Activation functions are essential to scale the outputs of neurons in all layers. During training, Neural Networks work to establish the underlying relationship between the input images and output labels [52]. This relationship is often complex and nonlinear, and so utilising activation functions that apply nonlinear functions on the output of each neuron layer is essential to correctly scale the outputs of all the neurons in the Neural Network.

More so optimizers are equally important as they use loss minimisation techniques to minimise the loss function, such as gradient descent. The optimizer's ability to function correctly is determined by the size of the steps taken during the optimization phase. The size of the steps taken is influenced by the learning rate, an equally important parameter to control [62].

The learning rate is set based on how much the weights and bias should change. Small changes in the weights are typically needed when the Neural Network is trying to learn small differences between data samples. This is achieved by using a small learning rate value. On the other hand, if the Neural Network model is learning very obvious differences in the data samples, then a higher learning rate is needed.

The batch size would also be changeable due to its importance when the size of the training and validation sets is in question [56, 57]. With small datasets, smaller batch sizes are recommended, while larger datasets can use bigger batches. The batch size has a direct effect on the number of steps in an epoch, which is derived by dividing the number of training samples by the batch size. A small database would also encourage more epochs since the model would not have many examples of what it is trying to learn.

The number of training images is vital to achieving high training and validation accuracy. The size of the image and its contents are also important, hence the need to control the size of the Convolutional Kernel. Convolutional Neural Networks (CNN) work by sliding a Convolution Kernel across the image, horizontally and vertically, calculating the dot product at each position. The dot product is between the kernel values and the pixel values covered by the Kernel. This is done to produce a weighted convolved feature, which hopefully represents a feature the CNN is trying to identify [35, 78]. Generally, useful features trying to be identified are small in comparison to the size of the image. Also, more than one

feature is being searched for, hence with these in mind, a small kernel size is used (typically 3x3). However, on some occasions, a 5x5 is used to minimise the CNN from picking up unwanted features. While there are other options to avoid this, hence it is crucial to have the option to change the size of the kernel.

The number of layers and number of neurons per layer directly affects the number of trainable parameters a Neural Network model has available during training. As more layers are added and more neurons per layer increases, so do the number of trainable parameters that are available for the Neural Network. These parameters are the weights that are learned and optimised by the Neural Network during stochastic gradient descent (explained in chapter three) [60, 62].

Typically, the more trainable parameters available, the more accurate the training will be. However, too many parameters and the Neural Network will learn unnecessary information regarding the data, resulting in overfitting [60]. For example, if the Neural Network is learning to identify a distracted behaviour, like a driver using the radio (the label), too many trainable parameters may cause the Neural Network to wrongly assume the colour of the driver's clothes as unique to identifying this particular action or label (overfitting). One way to prevent this from happening is by using a dropout layer (explained in chapter three) to reduce the number of parameters at the output layer. Hence, by controlling the number of layers and the number of neurons per layers, and the dropout ratio is essential to determine the number of trainable parameters made available to the Neural Network.

Lastly, to determine how the Neural Network model is performing a confusion matrix and classification report would be used to ascertain how well the model is doing for each label. Both the confusion matrix and classification report provide information on how each label is being classified by the Neural Network [73]. If the Neural Network model is struggling with identifying one label, it will impact the validation accuracy as a whole. The adaptive algorithm I proposed would be capable of compensating towards that label during training and skew the weights and biases in its favour, thereby improving the validation accuracy [85].

With the control of these parameters, the only inputs required for this proposed algorithm would be the directories for the training, validation and testing datasets.

Hence an adaptive algorithm would be capable of learning different datasets without user interference.

The aim would be to combine video classification with the adaptive algorithm discussed earlier in this chapter. The resultant algorithm would be able to use adaptive training and validation to improve their respective accuracies for classification in real-time.

The merits of Neural Networks show the potential this field has in furthering classification, detection and predictive systems. I hope the proposals, presented in this thesis, will help provide a foundation for further improvements to fatigue and distraction detection systems, thereby making travelling a safer endeavour.

6.3 References

[84] N. Yassine, S. Barker, K. Hayatleh, B. Choubey, and R. Nagulapalli, "Simulation of driver fatigue monitoring via blink rate detection, using 65 nm CMOS technology," *Analog Integrated Circuits and Signal Processing,* pp. 1-6, 2018.

[85] N. Lopes and B. Ribeiro, "Machine Learning for Adaptive Many-core Machines: A Practical Approach," 2015.

Chapter 7: Combined References

[1] ROSPA. "Driver Fatigue and Road Accidents Factsheet." https://www.rospa.com/media/documents/road-safety/driver-fatigue-factsheet.pdf (accessed February, 2020).

[2] R. Hunter, "Staying Awake, Staying Alive: The problem of fatigue in the transport sector," PACTS, London, 2013.

[3] "DaCoTA (2012) Fatigue," ed. Deliverable 4.8h of the EC FP7 Project DaCoTA.

[4] A. Williamson, "21 The Relationship between Driver Fatigue and Driver Distraction," *Driver distraction: Theory, effects, and mitigation,* p. 383, 2008.

[5] J. A. Stern, D. Boyer, and D. J. Schroeder, "Blink Rate As a Measure of Fatigue: A Review," Department of Psychology, Washington University, National Technical Information Service, 1994.

[6] W. Tansakul and P. Tangamchit, "Fatigue Driver Detection System Using a Combination of Blinking Rate and Driving Inactivity," *Journal of Automation and Control Engineering,* vol. 4, no. 1, 2016.

[7] "Optalert Adopts InterSystems Software to Prevent Fatigue-Related Industrial Accidents," ed. Sydney: InterSystms Cache and DeepSee Case Study, 2010.

[8] R. Murray Johns and R. Christopher Hocking, "Alertness sensing device," United States of America Patent 9,007,220, 2015.

[9] R. Ghoddoosian, M. Galib, and V. Athitsos, "A Realistic Dataset and Baseline Temporal Model for Early Drowsiness Detection," 2019, pp. 0-0.

[10] Y. Abouelnaga, H. M. Eraqi, and M. N. Moustafa, "Real-time distracted driver posture classification," *arXiv preprint arXiv:1706.09498,* 2017.

[11] H. M. Eraqi, Y. Abouelnaga, M. H. Saad, and M. N. Moustafa, "Driver distraction identification with an ensemble of Convolutional Neural Networks," *Journal of Advanced Transportation,* vol. 2019, 2019.

[12] I. Berezhnyy, G. N. Garcia Molina, and M. A. Weffers-Albu, "Detection system using photo-sensors " United States of America Patent 9,573,598, 2017.

[13] T. Mimar, "Driver distraction and drowsiness warning and sleepiness reduction for accident avoidance " United States of America Patent 9,460,601, 2016.

[14] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, 2001, vol. 1: IEEE, pp. I-I.

[15] M. Turk and A. Pentland, "Eigenfaces for recognition," *Journal of cognitive neuroscience,* vol. 3, no. 1, pp. 71-86, 1991.

[16] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau, "Real-time eye blink detection with GPU-based SIFT tracking," in *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, 2007: IEEE, pp. 481-487.

[17] "Cascade Classifier." The OpenCV Reference Manual. http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html (accessed July 2017).

[18] S. Soo, "Object detection using Haar-cascade Classifier," *Institute of Computer Science, University of Tartu,* pp. 1-12, 2014.

[19] R. Padilla, C. F. F. Costa Filho, and M. G. F. Costa, "Evaluation of haar cascade classifiers designed for face detection," *World Academy of Science, Engineering and Technology,* vol. 64, pp. 362-365, 2012.

[20] R. E. Schapire, "The Boosting Approach to Machine Learning An Overview," *Nonlinear Estimation and Classification,* 2001.

[21] G. Bradski, "The OpenCV Library," ed. Dr. Dobb's Journal of Software Tools: OpenCV, 2000.

[22] R. E. Schapire, "Explaining Adaboost," in *Empirical inference*: Springer, 2013, pp. 37-52.

[23] M. Yang, J. Crenshaw, B. Augustine, R. Mareachen, and Y. Wu, "AdaBoost-based face detection for embedded systems," *Computer Vision and Image Understanding,* vol. 114, no. 11, pp. 1116-1125, 2010.

[24] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman, "Eigenfaces vs. fisherfaces: Recognition using class specific linear projection," *IEEE Transactions on pattern analysis and machine intelligence,* vol. 19, no. 7, pp. 711-720, 1997.

[25] M. Çarıkçı and F. Özen, "A face recognition system based on eigenfaces method," *Procedia Technology,* vol. 1, pp. 118-123, 2012.

[26] M. Turk and A. Pentland, "Face recognition using eigenfaces," 1991, pp. 586-587.

[27] J. DeRuyck, "Detection of driver behaviors using in-vehicle systems and methods " United States of America Patent 9,714,037, 2017.

[28] M. E. Kramer, N. R. Lynam, D. P. O'Connell, and V. R. Nise, "Vision system for vehicle," United States of America Patent 9,632,590, 2017.

[29] N. R. Lynam, "Vision display system for vehicle " United States of America Patent 9,598,014, 2017.

[30] K. Nakamura and H. Takano, "Doze detection method and apparatus thereof " United States of America Patent 9,286,515, 2016.

[31] B.-J. Son, H.-I. Kim, and T.-H. Hong, "Apparatus and method of controlling mobile terminal based on analysis of user's face," United States of America Patent 9,239,617, 2016.

[32] B. Nilsson and E. Rosen, "Device and method for detecting drowsiness using eyelid movement " United States of America Patent 9,220,454 2015.

[33] Yang, Hsin-hsiang, and K. O. Prakah-Asante, "Driver drowsiness detection " Patent 9,205,844, 2015.

[34] K. R. Awad M, *Efficient Learning Machines*. Berkeley: Apress, 2015.

[35] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," *arXiv preprint arXiv:1511.08458,* 2015.

[36] M. Newman, *Networks: An Introduction*. OUP Oxford, 2010.

[37] X. Zhu and A. B. Goldberg, "Introduction to semi-supervised learning," *Synthesis lectures on artificial intelligence and machine learning,* vol. 3, no. 1, pp. 1-130, 2009.

[38] M. Wiering and M. van Otterlo, *Reinforcement Learning: State-of-the-Art*. Berlin: Springer, 2014.

[39] L. Dormehl. "What is an Artificial Neural Network? Here's everything you need to know." DIGITAL TRENDS. https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/ (accessed August 2019).

[40] J. A. Hertz, *Introduction to the theory of neural computation*. 2018.

[41] A. Sears-Collins. "Artificial Feedforward Neural Network With Backpropagation From Scratch." Automatic Addison. https://automaticaddison.com/artificial-feedforward-neural-network-with-backpropagation-from-scratch/ (accessed December 2019).

[42] V. Valkov. "Making a Predictive Keyboard using Recurrent Neural Networks — TensorFlow for Hackers (Part V)." medium. https://medium.com/@curiousily/making-a-predictive-keyboard-using-recurrent-neural-networks-tensorflow-for-hackers-part-v-3f238d824218 (accessed January 2020).

[43] H. Nam and B. Han, "Learning multi-domain convolutional neural networks for visual tracking.," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pp. 4293-4302, 2016.

[44] G. E. Hinton, *A Practical Guide to Training Restricted Boltzmann Machines*. Berlin: Springer, 2012.

[45] C. Nicholson. "A Beginner's Guide to Restricted Boltzmann Machines (RBMs)." pathmind. https://pathmind.com/wiki/restricted-boltzmann-machine (accessed January 2020).

[46] J. Torres, *First Contact with deep learning: Practical Introduction with Keras*. Watch This Space, 2018.

[47] S. Saha. "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way." Towards Data Science. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (accessed June 2019).

[48] M. Yani, "Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail," in *Journal of Physics: Conference Series*, 2019, vol. 1201, no. 1: IOP Publishing, p. 012052.

[49] J. Brownlee, *Deep Learning for Computer Vision* v1.7 ed.: Machine Learning Mastery, 2020.

[50] F. e. a. Chollet, "Keras," ed. GitHub repository: GitHub, 2015.

[51] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265-283.

[52] L. Vecci, F. Piazza, and A. Uncini, "Learning and approximation capabilities of adaptive spline activation function neural networks," *Neural Networks,* vol. 11, no. 2, pp. 259-270, 1998.

[53] T. Babs. "The Mathematics of Neural Networks." medium. https://medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05 (accessed December 2019).

[54] N. Kohl. "What is the role of the bias in neural networks?" Stack Overflow. https://stackoverflow.com/questions/2480650/what-is-the-role-of-the-bias-in-neural-networks (accessed May 2020).

[55] J. Brownlee, *Deep Learning with Python*, v1.18 ed.: Machine Learning Mastery, 2019.

[56] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1--learning rate, batch size, momentum, and weight decay," *arXiv preprint arXiv:1803.09820,* 2018.

[57] P. M. Radiuk, "Impact of training set batch size on the performance of Convolutional Neural Networks for diverse datasets," *Information Technology and Management Science,* vol. 20, no. 1, pp. 20-24, 2017.

[58] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700-4708.

[59] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research,* vol. 15, no. 1, pp. 1929-1958, 2014.

[60] S. Lawrence and C. L. Giles, "Overfitting and neural networks: conjugate gradient and backpropagation," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 2000, vol. 1: IEEE, pp. 114-119.

[61] Y. Zhao, J. Gao, and X. Yang, "A survey of Neural Network Ensembles," in *2005 International Conference on Neural Networks and Brain*, 2005, vol. 1: IEEE, pp. 438-442.

[62] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*: Springer, 2010, pp. 177-186.

[63] A. Lydia and S. Francis, "Adagrad-An Optimizer for Stochastic Gradient Descent," ed: May, 2019.

[64] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701,* 2012.

[65] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980,* 2014.

[66] A. Tato and R. Nkambou, "Improving adam optimizer," 2018.

[67] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," 2013: IEEE, pp. 8624-8628.

[68] G. Thimm, P. Moerland, and E. Fiesler, "The interchangeability of learning rate and gain in backpropagation neural networks," *Neural computation,* vol. 8, no. 2, pp. 451-460, 1996.

[69] R. Kurzweil, "The law of accelerating returns," in *Alan Turing: Life and legacy of a great thinker*: Springer, 2004, pp. 381-416.

[70] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: a structure for efficient numerical computation," *Computing in Science & Engineering,* vol. 13, no. 2, pp. 22-30, 2011.

[71] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006.

[72] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, 2010, vol. 445: Austin, TX, pp. 51-56.

[73] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *the Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[74] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in science & engineering,* vol. 9, no. 3, pp. 90-95, 2007.

[75] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks,* vol. 61, pp. 85-117, 2015.

[76] R. Vasudev. "Understanding and Calculating the number of Parameters in Convolution Neural Networks (CNNs)." https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-fc88790d530d (accessed 10 December 2019).

[77] Y. Zhang. "Number of Parameters in Dense and Convolutional Layers in Neural Networks." medium. https://medium.com/@zhang_yang/number-of-parameters-in-dense-and-convolutional-neural-networks-34b54c2ec349 (accessed August 2019).

[78] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285,* 2016.

[79] "State Farm Distracted Driver Detection." https://www.kaggle.com/c/state-farm-distracted-driver-detection/data (accessed August 2018).

[80] C. Sammut and G. I. Webb, *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[81] R. Garreta and G. Moncecchi, *Learning scikit-learn: machine learning in python*. Packt Publishing Ltd, 2013.

[82] P. R. Nicolas, *Scala for machine learning*. Packt Publishing Ltd, 2015.

[83] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375,* 2018.

[84] N. Yassine, S. Barker, K. Hayatleh, B. Choubey, and R. Nagulapalli, "Simulation of driver fatigue monitoring via blink rate detection, using 65 nm CMOS technology," *Analog Integrated Circuits and Signal Processing,* pp. 1-6, 2018.

[85] N. Lopes and B. Ribeiro, "Machine Learning for Adaptive Many-core Machines: A Practical Approach," 2015.

Chapter 8: Appendices

8.1 Types of Neural Networks.

Table. 8.1 Summary of different Neural Networks

Perception	Most basic form. Inputs connected to the output.
Feedforward	Nodes in each layer are connected to all nodes in the next layer. Has only one hidden layer.
Deep feedforward	Same as feedforward networks except that they have more than one hidden layer.
Autoencoder	Used for classification, clustering and feature compression. Can be trained without supervised learning. Supervised learning is when the number of neurones in each hidden layer equals or is more than the input layer.
Boltzmann Machine	All neurons are connected to one another. Some neurones are marked as inputs and made hidden. The input neurones become outputs when they receive the outputs from the rest of the neurones.
Restricted Boltzmann Machine	Can be trained using backpropagation. Backpropagation occurs only when data is passed back into the input layer once.
Deep belief network	Multiple Boltzmann Machines stacked together.
Extreme Learning Machine	Reduce the complexity of feedforward networks. Create sparse hidden layers with random connections. Require less computational power. Efficiency dependent on the task and input data set.
Recurrent Neural Network	The neurones in the hidden layers have their own output backpropagate into itself.
Echo State Network	A subtype of recurrent networks. Data passed to input. When being monitored for multiple iterations (recurrent feature), it is passed to the output. Weights in the hidden layer are updated.
Neural Turing Machine	A subtype of recurrent Neural Networks. These attempt to look at what is going on in the hidden layers.

8.2 Training and Validation on the Fatigue dataset.

8.2.1 Number of neurons in the dense layer

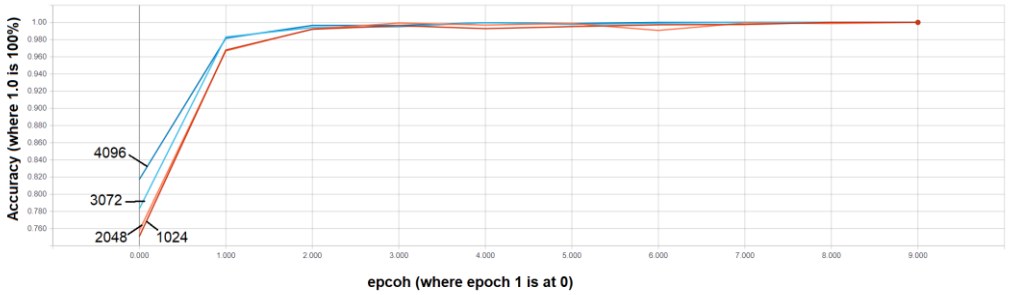

Fig. 8.1 Training Accuracy of the number of neurons in the dense layer for fatigue.

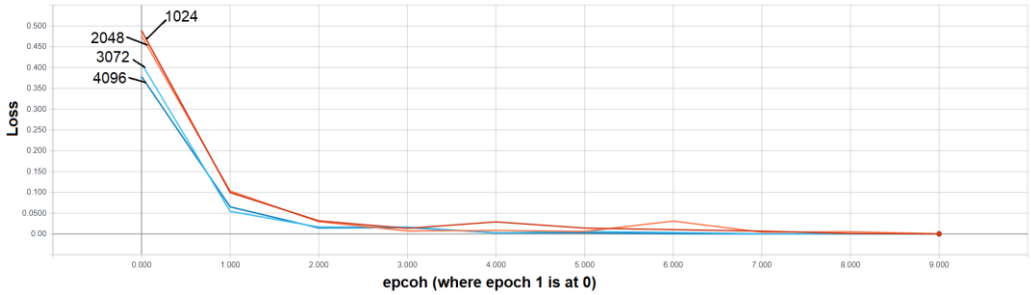

Fig. 8.2 Training Loss of the number of neurons in the dense layer for fatigue.

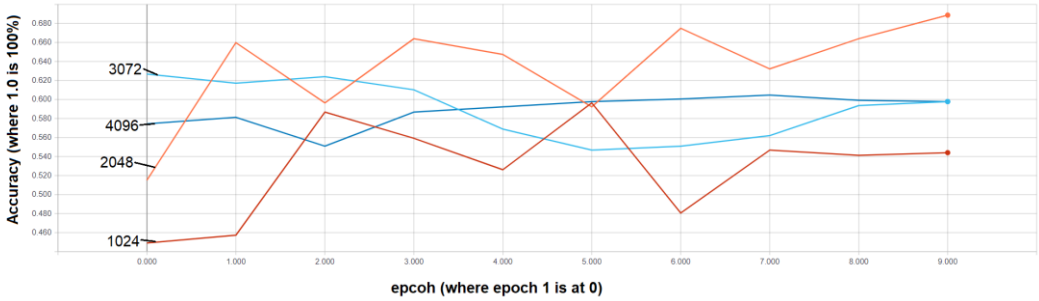

Fig. 8.3 Validation Accuracy of the number of neurons in the dense layer for fatigue.

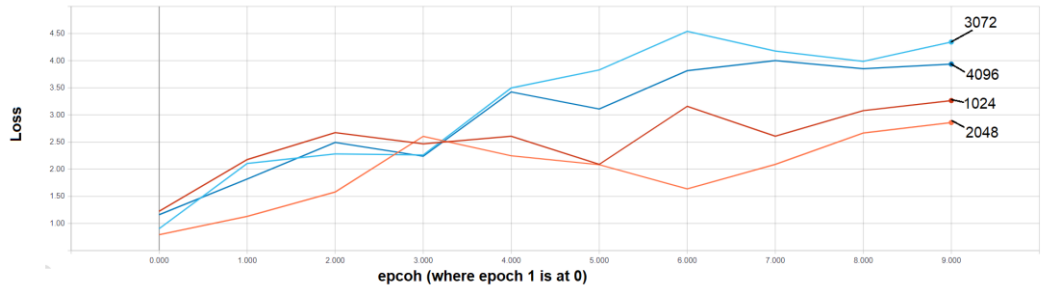

Fig. 8.4 Validation Loss of the number of neurons in the dense layer for fatigue.

8.2.2 Learning Rate

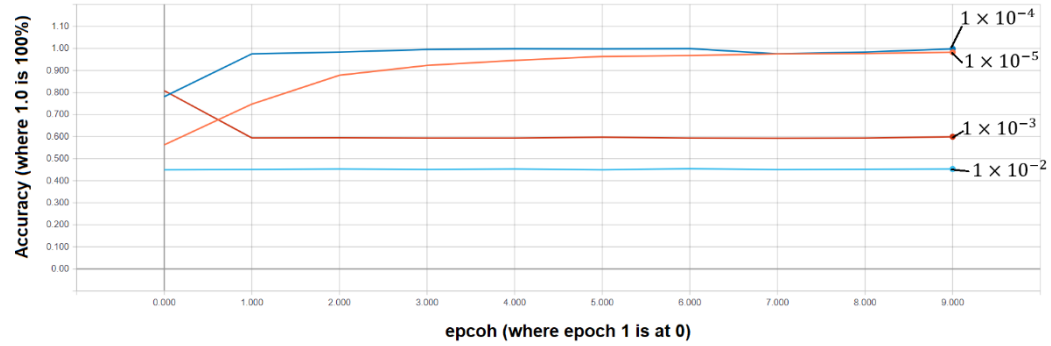

Fig. 8.5 Training accuracy for differing learning rates for fatigue

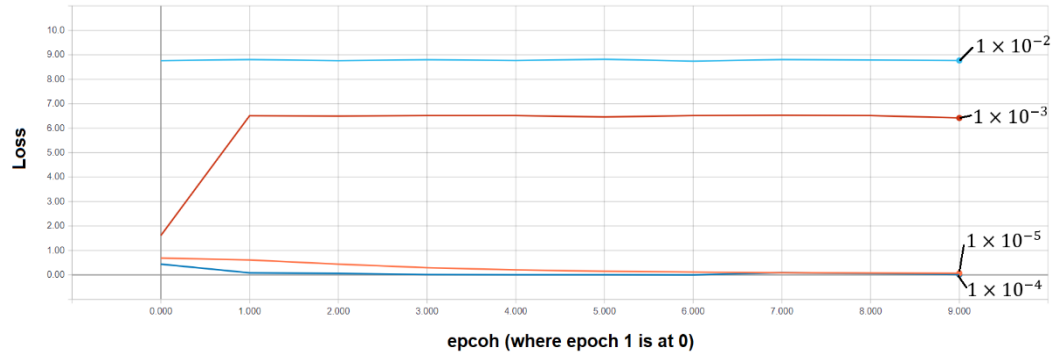

Fig. 8.6 Training Loss for differing learning rates for fatigue

Fig. 8.7 Validation Accuracy for differing learning rates for fatigue

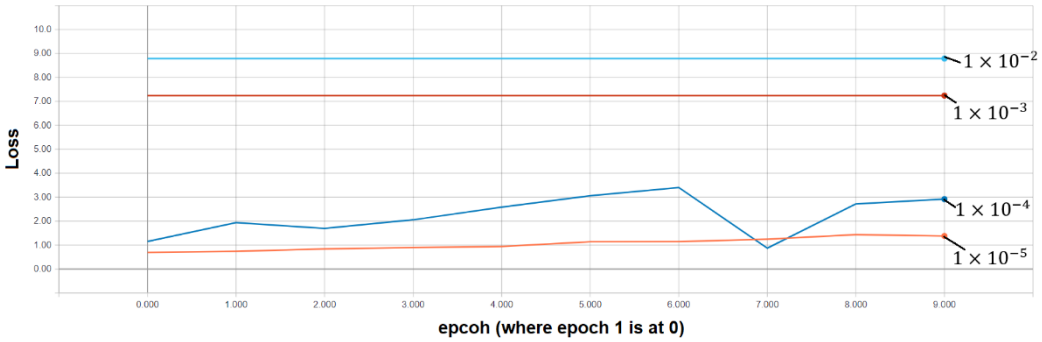

Fig. 8.8 Validation Loss for differing learning rates for fatigue

8.2.3 Optimizers

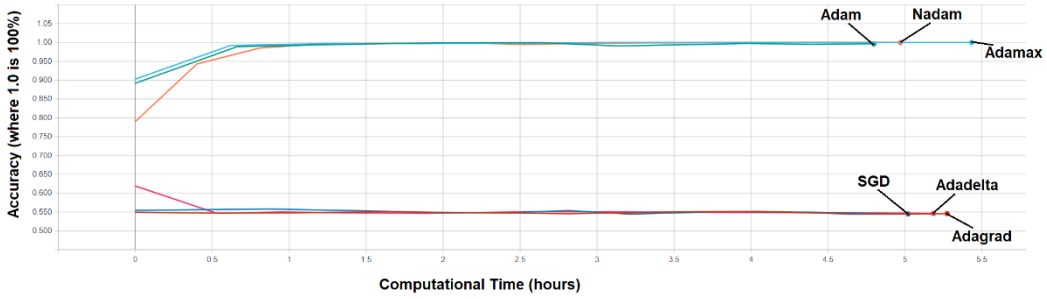

Fig. 8.9 Training accuracy for different optimizers for fatigue dataset.

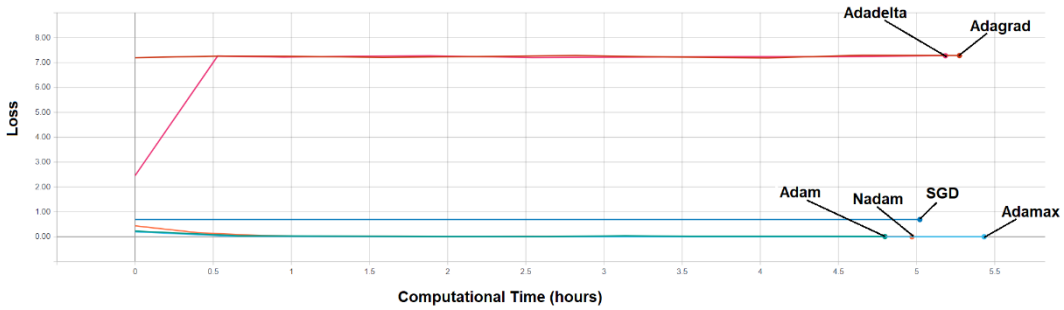

Fig. 8.10 Training Loss for different optimizers for fatigue dataset.

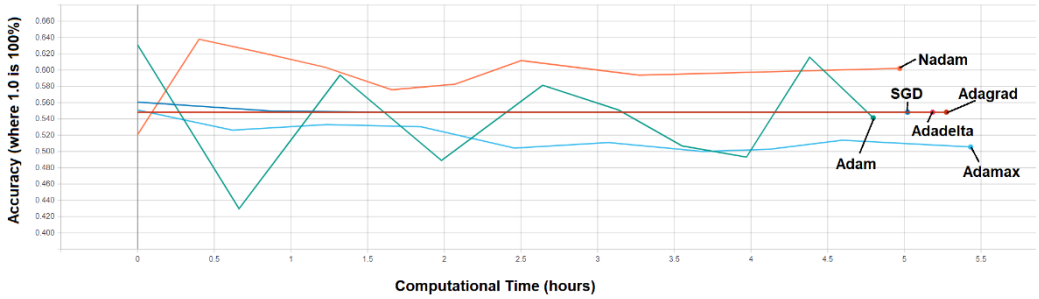

Fig. 8.11 Validation accuracy for different optimizers for fatigue dataset.

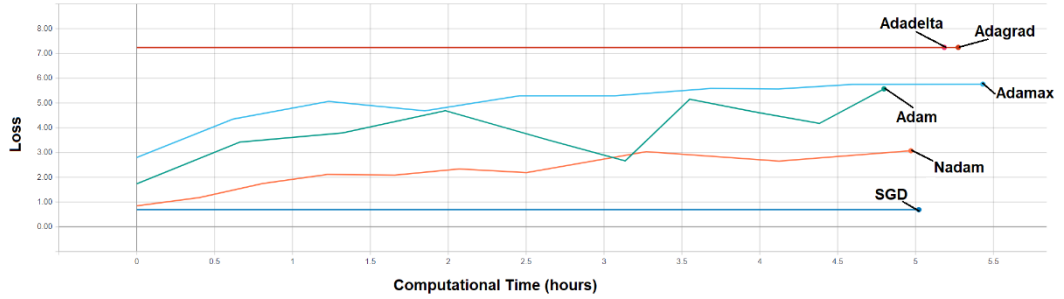

Fig. 8.12 Validation Loss for different optimizers for fatigue dataset.

8.3 Full code for the proposed algorithm

8.3.1 Libraries

```
# libraries used to produce the algorithm.
# Keras library used to form the architecture of the Neural Network model.
# matlplotlib used to create figures during testing.
# sklearn.metrics used to produce classification report and confusion matrix
import datetime
import time
import os
from PIL import Image
import numpy as np
import pandas as pd # imaging processing library
import tensorflow as tf
from scipy.ndimage._ni_label import _label
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib as mpl
import matplotlib.pyplot as plt
import keras
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model, model_from_json, Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Input, Dropout
from keras.optimizers import Adam, SGD, Nadam, Adadelta, Adagrad, Adamax
```

8.3.2 Training and Validation

```
### **Custom Functions**

class model_Configuration:
    def __init__(self, train_dir, valid_dir,test_dir, epoch_value,
                 batch_size_value, num_first_layer_filters, kernel_size,
                 num_dense_layer_neurons, no_labels, learning_rate):
        # variables commonly used
        self.epochs = epoch_value
        self.batch_size = batch_size_value
        self.num_first_layer_filters = num_first_layer_filters
        self.kernel_size = kernel_size
        self.num_dense_layer_neurons = num_dense_layer_neurons
        self.no_labels = no_labels
        self.learning_rate = learning_rate

        # minimum and maximum dimensions of the images in the dataset.
        self.maxwidth = 0
        self.maxheight = 0
        self.minwidth = 1920
        self.minheight = 1080
        self.image_counter = 0

        # resoltion of images in data set
        self.img_width_adjust  = 480

        # resoltion of images in data set
        self.img_height_adjust = 360
```

```
        # directories for training, validation and test datasets
        self.train_data_dir = train_dir
        self.valid_data_dir = valid_dir
        self.test_data_dir = test_dir

def Image_Dimensions(input_class):
# stores image dimensions in the class called input_class
# uses the PIL library (Image module)
# uses os library
# 2 inputs: dataset path and input dictionary
    image_directory = input_class.train_data_dir
    for subdir, dirs, files in os.walk(input_class.train_data_dir):
        for file in files:
            # check if appropiate file types in path location
            if file.endswith(".jpg") or file.endswith(".jpeg"):

                input_class.image_counter +=  1

                # filename of current file
                current_filename = os.path.join(subdir, file)
                current_image = Image.open(current_filename)

                # store the width and height of the image
                current_width, current_height = current_image.size

                # check if current dimesions are larger or smaller
                # than min and max dimensions.
                if current_width < input_class.minwidth:
                    input_class.minwidth = current_width

                if current_width > input_class.maxwidth:
                    input_class.maxwidth = current_width

                if current_height < input_class.minheight:
                    input_class.minheight = current_height

                if current_height > input_class.maxheight:
                    input_class.maxheight = current_height
    return

def plot_imgs(images_in_directory, rows = 3, cols =2):
# prints images in a plot of dimensions specified by rows and cols.
# if rows and cols not given, the default values are 3 and 2 respectively.
# takes three inputs: the directory where images are located and the dimensions of the plot.
    list_of_dirs = os.listdir(images_in_directory)
    for (pathname, dirname_list, filename_list) in os.walk(images_in_directory):
        idx = 1;

        if len(filename_list) > 0:
            # creates plot with dimensions proivded (or default values used)
            plt.figure(figsize=(rows, cols))

        # if the last file in the given directory is displayed then show image.
        for  file in filename_list :
            if file == "":
                plt.show()
                break

            #ensures full path name written correctly.
            full_path_name = pathname +  "/" + file
            plt.subplot(rows, cols, idx)
            idx = idx + 1 ;

            # show image obtained from the full path name
```

```
            img = plt.imread(full_path_name)
            plt.imshow(img)

            plt.tight_layout()

            # resets idx if all spaces on image (rows x cols) are filled.
            if idx > rows*cols:
                plt.show()
                idx = 1
                plt.figure(figsize=(rows, cols))
                print()

    return
##############################################################################
####################
def dataset_model_information(input_class, compiled_model):
# lists the directories in the given path and provides number of files in
# each subdirectory
    # empty list formed.
    # this will be the output after calling this function.
    directoy_count = []
    for subdir, dirs, files in os.walk(input_class.train_data_dir):
        # gives the number of files in the current directory and
        # stores them in input_class.image_counter.
        input_class.image_counter = len(files)
        # current directory (or sub directory stored in dirname)
        dirname = subdir
        # list appended with each set of location and file count.
        # output directoy_count is returned after calling this function.
        directoy_count.append((dirname, input_class.image_counter))


    categoryInfo = pd.DataFrame(directoy_count, columns=['Category', 'Count'])
    categoryInfo = categoryInfo.sort_values(by=['Category'])

    print(categoryInfo.to_string(index=False))
    print("Minimum Width:\t", input_class.minwidth, "\tMinimum Height:", input_class.minheight)
    print("Maximum Width:\t", input_class.maxwidth, "\tMaximum Height:", input_class.maxheight)
    print("Image Count:\t", input_class.image_counter)
    print(compiled_model.summary())

    return

##############################################################################
##############################
def dataset_setup(model_config):
    # ImageDataGenerator is used for augmentation of training data.
    # ImageDataGenerator takes a batch of images and augments them before they are sent
    # through the Neural Network.
    # helps ensure a more significant difference between validation data and training data.
    # ImageDataGenerator does not effect validation data. only determines split
    # not all options necessary. only which would be considered realistic.
    # horizontal flip realistic. changes side of the person from left to right and vice versa.
    # vertical flip not realistic. Would make things upside down.

    sample_data_gen = ImageDataGenerator(rescale=1. /255,   # 1. /255
                                         shear_range=0.2,     # 0.2
                                         zoom_range=0.2,      # 0.2
                                         horizontal_flip=True,
                                         validation_split=0.2)  # set validation split

    # create training dataset.
    # shuffle randomises images
    # color mode sets color of images.
```

```
    # class mode if multiple outputs use categorical
    # subset important when spliting validation data from same set.
    train_samples = sample_data_gen.flow_from_directory(
                                model_config.train_data_dir,
                                target_size=(model_config.img_width_adjust,
model_config.img_height_adjust),
                                shuffle=True,
                                color_mode = "rgb",
                                batch_size=model_config.batch_size,
                                class_mode='categorical',
                                subset='training')

    # create validation generator
    # shuffle important when distribution of data is in order
    # splitting data means validation gets last 10% not ideal, hence shuffle important.
    # same settigns as training generator
    valid_samples = sample_data_gen.flow_from_directory(
                                model_config.valid_data_dir,
                                target_size=(model_config.img_width_adjust,
model_config.img_height_adjust),
                                shuffle=True,
                                color_mode = "rgb",
                                batch_size=model_config.batch_size,
                                class_mode='categorical',
                                subset='validation')

    # create test dataset
    # useful if test datset present. not necesssary as validation is first stage of testing.
    test_samples = sample_data_gen.flow_from_directory(
                                model_config.test_data_dir,
                                target_size=(model_config.img_width_adjust,
model_config.img_height_adjust),
                                batch_size=model_config.batch_size,
                                class_mode='categorical')
    # Note uses training dataflow generator
    return train_samples, valid_samples, test_samples

def compile_model(input_class):
# function builds Neural Network model.
# takes 6 inputs and returns the compiled model.
# input_class has data required by input layer.
# 5 layers used.
    # input layer. Dimensions of images given as well as if the image is RGB or grayscale.
    # if RGB 3 used. If grayscale then one used.
    Input_layer = Input(shape=(input_class.img_width_adjust, input_class.img_height_adjust, 3),
name="input")

    # each convolutional layer followed immediately by a pooling layer.
    # convolutional layer connected to its pooling layer.
    # each pooling layer connected to the next convolutional layer.
    # Convolution 1
    conv1 = Conv2D(input_class.num_first_layer_filters, kernel_size=(input_class.kernel_size,
input_class.kernel_size), activation="relu", name="conv_1")(Input_layer)
    pool1 = MaxPooling2D(pool_size=(2, 2), name="pool_1")(conv1)

    # Convolution 2
    conv2 = Conv2D(input_class.num_first_layer_filters // 2, kernel_size=(input_class.kernel_size,
input_class.kernel_size), activation="relu", name="conv_2")(pool1)
    pool2 = MaxPooling2D(pool_size=(2, 2), name="pool_2")(conv2)

    # Convolution 3
    conv3 = Conv2D(input_class.num_first_layer_filters // 4, kernel_size=(input_class.kernel_size,
input_class.kernel_size), activation="relu", name="conv_3")(pool2)
    pool3 = MaxPooling2D(pool_size=(2, 2), name="pool_3")(conv3)
```

```
    # Convolution 4
    conv4 = Conv2D(input_class.num_first_layer_filters // 8, kernel_size=(input_class.kernel_size,
input_class.kernel_size), activation="relu", name="conv_4")(pool3)
    pool4 = MaxPooling2D(pool_size=(2, 2), name="pool_4")(conv4)


    # Convolution 5
    conv5 = Conv2D(input_class.num_first_layer_filters // 16, kernel_size=(input_class.kernel_size,
input_class.kernel_size), activation="relu", name="conv_5")(pool4)
    pool5 = MaxPooling2D(pool_size=(2, 2), name="pool_5")(conv5)


    # Fully Connected Layer Dense. Flatten function required before Dense.
    flatten = Flatten()(pool5)
    fc1 = Dense(input_class.num_dense_layer_neurons, activation="relu", name="fc_1")(flatten)

    # output layer also dense. It has ten neurons which are the labels.
    output_layer = Dense(input_class.no_labels, activation="softmax", name="softmax-output")(fc1)

    # finalize and compile
    # loss different depedning on number of nabels at output.
    # if 2 labels then binary_crossentropy. if more than 2 then categorical_crossentropy
    compiled_model = Model(inputs=Input_layer, outputs=output_layer)
    compiled_model.compile(optimizer=Adam(input_class.learning_rate),
loss='categorical_crossentropy', metrics=["accuracy"])

    return compiled_model

def fit_model(model, train_generator, valid_generator, batch_size, epochs):
    # the checkpoint callback is used for tracking the validation accuracy to save the best model
    # only the best weights are saved that produce the best validation accuracy.
    checkpoint = ModelCheckpoint(SAVED_MODEL_NAME, monitor='val_acc', verbose=1,
save_best_only=True, mode='max')

    # the tensorboard callback is used for logging the losses and accuracies for the model
    # saves data for both trainig and validation.
    # also displays the model diagram.
    tbCallback = TensorBoard(log_dir=log_dir, histogram_freq=0, write_graph=True,
write_images=True)

    # trains the model
    # uses data stored in dictionary.
    # callbaks make sure correct saving is done of the weights and accuracies after each epoch.
    model.fit_generator(
        train_generator,
        steps_per_epoch=train_generator.samples // batch_size,
        epochs=epochs,
        validation_data=valid_generator,
        validation_steps=valid_generator.samples // batch_size,
        verbose=1, callbacks=[checkpoint, tbCallback] )


    return model

### **setup**

np.random.seed(42)

# name model and its weights saved under
model_version = "model_name"
epoch = 5
batch_size = 32
learning_rate = 0.001
```

```
num_first_layer_filters=128
kernel_size=3
num_dense_layer_neurons=1024
no_labels = 10


# naming the weights and models with the variable (and its assigned value)
# that is being investigated.
SAVED_WEIGHTS_NAME = "myweights_" + model_version + ".h5"
SAVED_MODEL_NAME   = "mymodel_" + model_version + ".json"



SELECTION = 1

if SELECTION == 1 :
    TRAIN_DATA_DIRECTORY = '/content/drive/My Drive/Colab
Notebooks/AUC\v1_cam1_no_split/train'
    VALID_DATA_DIRECTORY = '/content/drive/My Drive/Colab
Notebooks/AUC\v1_cam1_no_split/train'
    TEST_DATA_DIRECTORY = '/content/drive/My Drive/Colab
Notebooks/AUC\v1_cam1_no_split/train'

elif SELECTION == 2 :
    TRAIN_DATA_DIRECTORY = '/content/drive/My Drive/Colab
Notebooks/AUC\v1_cam1_no_split/train'
    VALID_DATA_DIRECTORY = '/content/drive/My Drive/Colab
Notebooks/AUC\v1_cam1_no_split/train'
    TEST_DATA_DIRECTORY = '/content/drive/My Drive/Colab
Notebooks/AUC\v1_cam1_no_split/train'


### **Generate Data**
# model_configuration called
model_info = model_Configuration(TRAIN_DATA_DIRECTORY, VALID_DATA_DIRECTORY,
                                 TEST_DATA_DIRECTORY, epoch, batch_size,
                                 num_first_layer_filters, kernel_size,
                                 num_dense_layer_neurons, no_labels, learning_rate)

# Dataset_setup called
# data generated. traiing and validation split 80% and 20% respectively.
train_generator, valid_generator, test_generator = Dataset_setup(model_info)



### **Build and Save Model**
# compile model
my_compiled_model = compile_model(model_info)

# displayn model summary and dataset information
dataset_model_information(model_info, my_compiled_model)


# serialize model to JSON
mysaved_model = my_compiled_model.to_json()
with open(SAVED_MODEL_NAME, "w") as json_file:
    json_file.write(mysaved_model)

# save location for model.
save_dir_tf = '/content/drive/My Drive/Colab Notebooks'
+ datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
+ model_version

# At the beginning save the  first weights
```

```
# Only save weights if the current val_acc is better than the val_acc
# saved file weights
checkpoint = ModelCheckpoint(SAVED_WEIGHTS_NAME, monitor='val_acc',
                             verbose=1,
                             save_best_only=True,
                             mode='max')

### **TRAINING AND VALIDATION**
# the checkpoint callback is used for tracking the validation accuracy to
# save the best weights.
tbCallback = TensorBoard(log_dir=save_dir_tf,
                         histogram_freq=0,
                         write_graph=True,
                         write_images=True)

# fit_generator trains the compiled model.
# checkpoint for saving weights  that give the best validation accuracy.
# saved weights are saved in a .h5 file.
# tb callback for saving training losses and accuracies
my_compiled_model.fit_generator(train_generator,
                                steps_per_epoch = train_generator.samples
                                        // model_info.batch_size,
                                epochs = model_info.epochs,
                                validation_data = valid_generator,
                                validation_steps= valid_generator.samples
                                        // model_info.batch_size,
                                verbose=1,
                                callbacks=[checkpoint, tbCallback] )
```

8.3.3 Evaluation

```
### **Evaluation**
print("############### Begin Evaluation ########################")
#valid_generator is the validation dataset created with the function: setup_data
#valid_generator.samples is from validation_steps in the function: fit_generator
# batch_size - variable loacted at begining.
scores = my_compiled_model.evaluate_generator(valid_generator,
                                              steps=valid_generator.samples
                                              // batch_size)
print("Evaluation of compiled model with weights loaded.")
print("Loss: " + str(scores[0]) + " Accuracy: " + str(scores[1]))

# Scikit-learn version of the Confution Matrix and Classification Report
# floor division -> "num_of_test_samples // config.batch_size + 1"
# batch size for training and validation is the same value  = 16.
# resetting generator. Important if valid_generator used for prediction
valid_generator.reset()

prediction_matrix = my_compiled_model.predict_generator(valid_generator,
                                                        steps=valid_generator.samples
                                                        // batch_size +1)
# np.argmax gives the maximum indicies in prediction_matrix
max_pred_value = np.argmax(prediction_matrix, axis=1)

#calculate and print confusion matrix and classifaction report.
print(' Scikit-learn Confusion Matrix')
print(confusion_matrix(valid_generator.classes, max_pred_value))
print('Scikit-learn Classification Report')
print(classification_report(valid_generator.classes, max_pred_value))
```

8.3.4 Testing

```
# testing intelligence of model. Hasn't seen these images
# testing file.
IMAGES_PATH = '../_DataSet_/AUC\v1_cam1_no_split/test/'

# Load model and set path directory to save the result.
MODEL_PATH = 'model_name.h5'
SAVE_RESULTS_DIR = './result'
# dictionary labels made of key and value.
# These labels are for AUC dataset
MAPPING_DICTIONARY = {0: "Safe Driving",
                      1: "Texting Right",
                      2: "Talking on the Phone - Right",
                      3: "Texting Left",
                      4: "Talking on the Phone - Left",
                      5: "Operating the Radio",
                      6: "Drinking",
                      7: "Reaching Behind",
                      8: "Hair and Makeup",
                      9: "Talking to Passenger"}
# resize images
def Resize_Image_for_CNN(image, width, height):
    resized_image = cv2.resize(image, (width, height))
    resized_image = resized_image.astype('float32')
    resized_image /= 255.

    # expand the dimensions of the image
    resized_image = np.expand_dims(resized_image, axis=0)
    return resized_image

# variables initialized
rows = 5; cols = 5; f = []; idx = 1
for (dirpath, dirnames, filenames) in walk(IMAGES_PATH):
    f.extend(filenames)
    break
# load the model by using path of the model
cnn_model = load_model(MODEL_PATH)
# create list of images and store in imageFileNameList
imageFileNameList = list()
imageNameList = [ f for f in os.listdir(IMAGES_PATH) ]
imageFileNameList += [os.path.join(IMAGES_PATH, file)
                      for file in imageNameList]
# create plot of figure size rows x cols
plt.figure(figsize=(rows, cols))
# test model on images one at a time. loops through imageFileNameList
for imageFileName in imageFileNameList :
    image = cv2.imread(imageFileName)
    cv2.imshow('out', image)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    cv2.imshow('out', image)
    input_cnn = Resize_Image_for_CNN(image)
    # winner_class assigns image a label from dictionary
    cnn_output = cnn_model.predict(input_cnn)
    winner_class = cnn_output[0].argmax()
    # print label on image
    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(image, MAPPING_DICTIONARY[winner_class],
              (10, 30),font, 1, (0, 0, 255), 1, cv2.LINE_AA)
    cv2.imwrite('test.png', image)
    cv2.imshow('out', image)
    # Only executes if there is a free plot space to inset image
    # if no free plot space then new plot is created.
```

```
    if idx <= 25:
        full_path_name =  'test.png'
        plt.subplot(5, 5, idx)
        idx = idx + 1 ;
        img = plt.imread(full_path_name)
        plt.imshow(img)
        plt.tight_layout()
cv2.waitKey(0) # waits for user input
```

Chapter 9: Published Papers

1. Yassine, N., Barker, S., Hayatleh, K. et al. Simulation of driver fatigue monitoring via blink rate detection, using 65 nm CMOS technology. Analog Integrated Circuits and Signal Processing 95, 409–414 (2018). https://doi.org/10.1007/s10470-018-1151-3.

The above article has been removed from this version
of the thesis due to copyright restrictions