Design and implementation of a convolutional neural network to classify pecan nut cultivars in a post-harvest application

by

Johann Daniël Joubert

A thesis submitted in the fulfilment for the degree
Master of Engineering: Electrical Engineering

in the Faculty of Engineering and the Built Environment

at the Cape Peninsula University of Technology

Supervisor: Prof Mohammed Kahn

Bellville

November 2020

CPUT copyright information

The thesis may not be published either in part (in scholarly, scientific or technical journals), or as a whole (as a monograph), unless permission has been obtained from the University

DECLARATION

I, Johann Daniël Joubert, declare that the contents of this research thesis represent my own unaided work and that the research thesis has not previously submitted for academic examination towards any qualification. Furthermore, it represents the authors own opinions and not necessarily those of the Cape Peninsula University of Technology.

____________________   ____13/09/2020_______

Signed   Date

ABSTRACT

Roughly 85%-90% of the 14 000 tons of pecan nuts produced in South Africa is exported to the international market. This makes South Africa one of the four biggest exporters of pecan nuts in the world. Market survey reports indicate that the demand for pecan nuts globally is on the rise, and for that reason, South African farmers should invest in better technology to stay competitive while keeping up with the demand. The application of convolutional neural networks (CNN) has successfully applied in various domains, and recently entered also the domain of agriculture. Although not new, recent improvements and access to better tools for image processing and data analysis problem are delivering promising results.

In this research, an overview is presented of current commercial sorting technology and applications where machine learning is already being researched. The application to pecan nuts is novel in the sense that there are to the author's knowledge no other studies which applied a convolutional neural network to classify pecan nut cultivars.

This study laid a foundation for future research into this field by generating a dataset of over 3000 pecan nut images of three cultivars and by determined that by making use of low-cost cameras and hardware an excellent classification accuracy of 98% could be achieved. The research implemented a transfer learning process on a VGG16 and MobileNetV2 model and compared the results of both models. Other key visual parameters, such as size and colour, are also extracted and presented for future research in the field.

Keywords: Convolutional Neural Network, Support vector machine, Pecan nut, Agriculture, Food safety machine vision inspection, Pattern recognition, Machine learning

ACKNOWLEDGEMENTS

I would first like to thank our Heavenly Father for granting me the opportunity, wisdom and determination I needed to fulfil a life goal of mine and complete a master degree in engineering.

A massive thank goes to my wife, without your support and endless love, this would not have been possible. Then to my new-born son, may you come to understand that with hard work and persistence, anything is possible.

Then to Prof Khan for his support and guidance throughout my research.

TABLE OF CONTENTS

DECLARATION ii
ABSTRACT iii
ACKNOWLEDGEMENTS iv
TABLE OF CONTENTS v
LIST OF FIGURES viii
LIST OF TABLES ix
LIST OF ABBREVIATIONS x

Chapter 1: Introduction 1

1.1. Background to the research problem 1
1.2. Statement of the research problem 4
1.3. Research questions 4
1.4. Investigative questions 4
1.5. Project objective 4
1.6. Delineation of the research 5
1.7. The significance of the research 5
1.8. Thesis outline 6

Chapter 2: Related work 7

2.1. Overview of commercial optical sorting machines 7
2.1.1. Feed systems 8
2.1.2. Optics 8
2.1.3. Ejection process 8
2.1.4. Image processing algorithms 8
2.2. Current work in the field 9
2.2.1. Data acquisition 10
2.2.2. Data pre-processing 10
2.2.3. Feature extraction 11
2.2.4. Classification 12
2.3. Work-related to pecan nuts 12
2.1. Summary 13

Chapter 3:Research methodology 14

3.1.1. WP101,WP201 Capturing images 14
3.1.2. WP103 and WP104 Implement and test model 17
3.1.3. WP202 and WP203 Improve model 20

3.1.1. WP204 Validate model on hardware 20
3.1.2. WP401 and WP402 20
3.2. Summary 20
Chapter 4: A brief introduction to neural networks 21
4.1. Activation functions : 25
4.2. Backprogration algorithm 27
4.2.1. Forward pass: 28
4.2.2. Backward pass: 28
4.3. Summary 31
Chapter 5: Convolutional Neural network 32
5.1. Common architectures 33
5.1.1. Visual Geometry Group Network (VGGNet) 33
5.1.2. MobilenetV2 34
5.2. Understanding convolutions: 37
5.3. Layer types : 40
5.3.1. Convolutional layer (CONV) 40
5.3.2. Activation (ACT) 41
5.3.3. Pooling (POOL) 42
5.3.4. Fully-connected (FC) 42
5.4. Loss functions : 43
5.4.1. Mean Square Error (MSE) : 43
5.4.2. Cross-Entropy : 43
5.5. Optimisation algorithms 44
5.5.1. Gradient descent 44
5.5.2. Stochastic Gradient Descent (SGD) 45
5.6. Regularisation approaches 45
5.6.1. L2 Regularisation 45
5.6.2. Data augmentation 45
5.6.3. Dropout 45
5.6.4. Early stopping 46
5.7. Invariance 46
5.7.1. Rotation invariance 46
5.7.2. Scale invariance 46
5.7.3. Translation invariance 46
5.8. Hierarchical feature learning 47

5.9. Training methods 48
5.9.1. From Scratch: 48
5.9.2. Transfer learning: 49
5.9.2.1. Feature extracting 49
5.9.2.2. Fine-tuning 50
5.10. Summary 50
Chapter 6: Implementation of a Convolutional Neural Network. 51
6.1. Hardware implementation 51
6.2. Data capturing and data pre-processing 54
6.2.1. Data pre-processing 56
6.3. Software implementation 59
6.3.1. Training VGG16 60
6.3.2. Training MobileNetV2 76
6.3.3. Results 80
6.3.3.1. Classification 80
6.3.3.2. Size measurements 85
6.3.3.3. Ratio measurements 86
6.3.3.4. Colour measurements 89
6.4. Summary 91
Chapter 7: 92
7.1. Conclusions 92
7.2. Recommendations 95
References 96
Appendix A 99

LIST OF FIGURES

Figure 1-1 World pecan nut production and price trends (Farmer's Weekly, 2018, p. 35) 1
Figure 1-2 SA pecan nut production (In Shell) & price trends (Farmer's Weekly, 2018, p. 35). 1
Figure 2-1 Components and layout of a typical sorting machine(Guggisberg and Bosset, 2003, p. 116) 7
Figure 2-2 Process of fruit and vegetable classification 10
Figure 2-3 Fruit images segmentation techniques (Bhargava and Bansal, 2018, p. 5) 11
Figure 2-4 Efficiency for quality analysis of fruits and vegetables based on colour features. (Bhargava and Bansal, 2018, p. 6)........ 11
Figure 2-5 Efficiency for quality analysis of fruits and vegetables based on classification techniques (Bhargava and Bansal, 2018, p. 11)........ 12
Figure 3-1 Work package breakdown 14
Figure 3-2 Types of ML algorithms 15
Figure 4-1 Overview AI, Machine and Deep Learning inspired by (Collet, 2018, p4)........ 21
Figure 4-2 Machine Learning a new paradigm (Collet, 2018, p5) 22
Figure 4-3 Biological Neuron with the inspired mathematical model........ 22
Figure 4-4 A Perceptron neuron. 23
Figure 4-5 Multilayer feedforward Network 24
Figure 4-6 Sigmoid activation function 26
Figure 4-7 ReLU activation function 26
Figure 4-8 Forward Pass........ 27
Figure 4-9 Backward Pass 29
Figure 5-1 VGG16 Architecture (Loukadakis, Cano and O 'boyle, 2018) 33
Figure 5-2 MobilenetV1 Architecture ((Howard et al., 2017)........ 34
Figure 5-3 Depth-wise Separable Convolution (https://towardsdatascience.com/deep-dive-into-the-computer-vision-world-f35cd7349e16) 35
Figure 5-4 MobileNetV2 architecture (Sandler et al., 2018) 36
Figure 5-5 Convolve operation no padding LEFT: Kernel, Middle: Original matrix, Right: Output matrix 37
Figure 5-6 Convolve operation zero-padding LEFT: Kernel, Middle: Original matrix with zero padding, Right: Output matrix 38
Figure 5-7 Blur and Edge detection with convolution 39
Figure 5-8 Activation Map (ROSEBROCK, 2017,p182) 40
Figure 5-9 Max Pooling operation with different stride length (http://cs231n.stanford.edu/slides/2016/winter1516_lecture7.pdf) 42
Figure 5-10 Gradient Descent (https://www.coursera.org/learn/machine-learning) 44
Figure 5-11 Traditional Feature Creation vs Deep Learning(ROSEBROCK, 2017)........ 47
Figure 6-2 Classification Process flow 51
Figure 6-3 Hardware implementation overview 52
Figure 6-4 Actual Hardware 53
Figure 6-6 Software and hardware stack (Collet, 2018, p62) 19
Figure 6-7 VGG16 Convolutional base 61
Figure 6-8 VGG16 Model architecture........ 69
Figure 6-9 VGG16 Classification layer 71
Figure 6-10 VGG16 Training and Validation 73
Figure 6-11 VGG16 Fine-tuning training parameters 74
Figure 6-12 MobileNetV2 Convolutional base........ 76

Figure 6-13 MobileNetV2 Convolutional base parameters 77
Figure 6-14 MobileNetV2 Classifier added 78
Figure 6-15 MobileNetV2 Convolutional base freeze 78
Figure 6-16 MobileNetV2 Training and Validation 79
Figure 6-17 MobileNetV2 Fine-tuning training parameters 80
Figure 6-18 Confusion Matrix 81
Figure 6-19 VGG16 Layer activation 83
Figure 6-20 Size measurements 85
Figure 6-21 Length to Height ratios of samples 87
Figure 6-22 Mahan colour spectrum 89
Figure 6-23 Shoshoni colour spectrum 89
Figure 6-24 Wichita colour spectrum 90
Figure 6-25 Combined Average and Maximum colour spectrum 90

LIST OF TABLES

Table 2-1 Comparison between automatic sorting and human inspection(Toyofuku, Haff and Pearson, 2013, p. 237) 9
Table 6-1 Mahan, Shoshoni and Wichita pecan nuts 55
Table 6-2 Rotation of pecan nut 56
Table 6-3 Remove background process 57
Table 6-4 Dataset 64
Table 6-5 Data augmentation 65
Table 6-6 Number of parameters and memory requirements 82
Table 6-7 VGG16 Grad-CAM 84
Table 6-8 MobileNetV2 Grad-CAM 84

LIST OF ABBREVIATIONS

ANN	Artificial neural networks
BPNN	Back-Propagation Neural Network
CT	computed tomography
CCD	Charge-coupled device
CNN	Convolution Neural Network
CMOS	Complementary metal–oxide–semiconductor
COTS	Commercial off-the-shelf,
CPUT	Cape Peninsula University of Technology
Kg	Kilogram
KNN	K-Nearest Neighbours
MRI	Magnetic resonance imaging
nm	Nanometre
PCA	Principal component analysis
SVM	Support-vector machine
X-Ray	X-radiation

Chapter 1: Introduction

1.1. Background to the research problem

South Africa is one of the four biggest pecan nut producing countries in the world. The other 3 are the US, Mexico and Australia. In 2017 85% to 90% of the roughly 14 000t of good quality pecan nuts were exported to China(Farmer's Weekly, 2018, p. 34). In the Asia market, the demand for pecan nuts is increasing in line with the population growth, whereby the consumers are becoming more health-conscious and favouring a healthier snack. Pecan nuts are a rich source of phytochemicals with antioxidant, antiproliferative, anti-inflammatory, antiviral properties. The nuts contain mono- and polyunsaturated fatty acids, and regular pecan consumption has been credited to decrease total cholesterol and LDL cholesterol levels, lower the risk of heart disease (A. A. Gardea and M. A. Martínez-Téllez, Development, Mexico and E. M. Yahia and Queretaro, 2011, p. 162).

According to the ABSA Agricultural Outlook Spring edition 2017/2018, South Africa can expect to see growth in line with the increased global demand as seen in Figure 1-1. Figure 1-2 shows the expected price increase with demand until 2021 (Farmer's Weekly, 2018, p. 35).

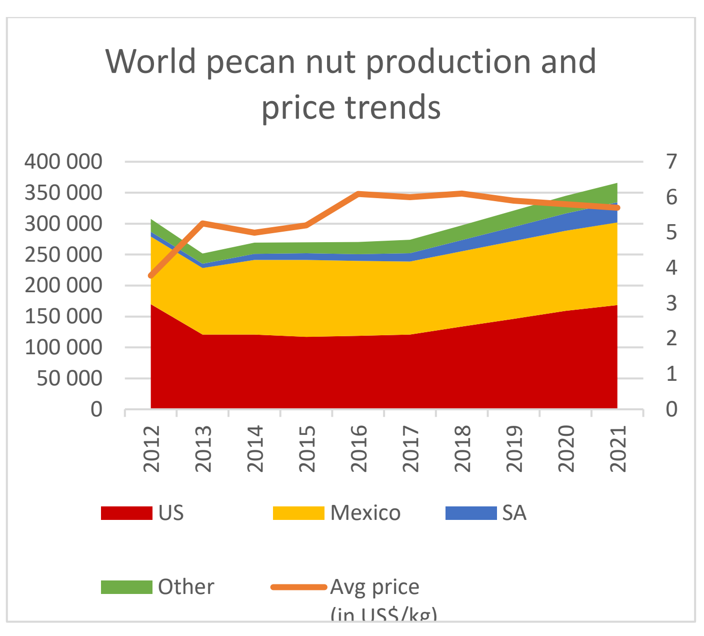


Figure 1-1 World pecan nut production and price trends (Farmer's Weekly, 2018, p. 35)

Figure 1-2 SA pecan nut production (In Shell) & price trends (Farmer's Weekly, 2018, p. 35).

Figure 1-1 shows the world biggest pecan nut producers and the average price trend. Although with a relatively small market percentage, The production of pecan nuts in South Africa is expected to significantly increase in the next few years as more orchards are coming into production, however certain inflation factors like the current depreciating exchange rate are putting more pressure on the agricultural sector as a whole, as certain vital inputs as fuel and fertiliser are expected to increase over the next ten years(BFAB, 2018, p. 10).

Figure 1-2 shows the local production vs export market. As seen above the majority of what is produced are exported, and the local market consumes only a small percentage. As the export demand increased the price per kilogram increased from 2017-2020, but are slowly slowing down, which correspond well to the research on the internal market.

There are various reason why a grower would have different cultivars in an orchard, one of the main reasons is the pecan nut tree produced best if pollinated by another pecan variety versus self-pollination which leads to poor nut growth and seldom produce large crops. Other reasons would be to mitigate the risk of funguses which is a severe challenge in humid climates. Different cultivars also bear fruit at different stages is beneficial if a specific geographic region or market such as Thanksgiving or Christmas holiday season is targeted (Lenny Wells and Patrick Conner, 2015).

The harvest needs to be cleaned and sorted according to size before export. Different cultivars contain a different volume of the kernel which in turns yields a different price. A Shosoni nut typically has 53% kernel ratio, A Mahan has 58%, and the Wichita has 62% (L. J. Grauke and T. E. Thompson, no date).

There is then a benefit to sort the pecan nuts by cultivar as the kernel percentage contributes significantly to the weight of the nut which determines the price per KG. Different pecan nut cultivars also yield different harvests each year, for that reason, a farmer wants to diverse the cultivars in the orchard to achieve a constant harvest every year (L. J. Grauke and T. E. Thompson, no date).

Automatic sorting machines are available and been adopted worldwide, but regions with low labour cost manual sorting of food products with the human eye and hand is still widely practised(Guggisberg and Bosset, 2003, p. 115).

Toyofuku et al. state the two main challenges with manual sorting are: defects too small or subtle for the human eye to detect and the required volume and speed at which the product needs to be inspected. Automatic systems can reliably and consistently inspect items as small as individual grains of wheat and remove the undesired product in real-time. With advances in technology devices such as sensors and imaging capture devices, the implementation cost has been lowered and enabled automatic sorting of a wide range of nuts (Toyofuku, Haff and Pearson, 2013, p. 231).

Cooperative associations are formed by individual growers to make a more efficient investment into facilities and equipment when they individually do not have the resources or volume of produce for a favourable cost-benefit ratio to own their equipment. Otherwise, the individual grower will sell the crop to processors at a reduced cost without the need for further capital investment(A. A. Gardea and M. A. Martínez-Téllez, Development, Mexico and E. M. Yahia and Queretaro, 2011, p. 154).

By providing an innovative integrated low-cost solution to the market, world-class technology can be accessible by the individual growers and according to the BFAB Agricultural Outlook 2017-2028 growers need to be productive and invest in the best technology to be sustainable in a fast-growing sector to achieve success in this competitive market. (BFAB, 2018, p. 16)

1.2. Statement of the research problem

Post-harvest classification of pecan nuts is a timeous, expensive and error-prone process. A Commercial sorting machine drastically reduces the time and errors made during a post-harvest sorting process. However, this equipment is typically used only by processors as the cost is prohibitively expensive for the small individual grower of pecan nuts.

1.3. Research questions

What accuracy can be achieved by using commercial off the shelve low-cost hardware and opensource software to classify pecan nut cultivars?

1.4. Investigative questions

The following investigation questions will be used to guide the research:

1. What accuracy can be achieved by using a low accuracy camera and lens?
2. Can transfer learning be used to retrain a CNN successfully on pecan nuts?
3. What type of pre-processing would improve accuracy?
4. What are other features available in the images?

1.5. Project objective

The objectives of this research project are to:

- To establish a suitable camera set up to capture the pecan nut dataset by conducting experiments.
- To capture a dataset of pecan nut images of three cultivars, which will be used to train a convolution neural network.
- To implement a convolution neural network based on two different architectures (MobilenetV2 and VGG16).
- Determine what accuracy can be achieved by utilising machine learning methods in classifying between different pecan nut cultivars.
- Determine if by using machine learning methods, a low-cost hardware solution could be developed.

1.6. Delineation of the research

The following delimitations have been set for this project:

- 3 Pecan nut cultivars will be used in the research.
- The research will focus on implementing a CNN classifier.
- Commercial Off the Shelf (COTS) hardware will be used.

1.7. The significance of the research

To help and improve the South African pecan nut industry to be more cost-effective and productive by improving the following:

- To lower the cost of the automated classification process after harvest.
- To empower the producer/grower to inspect the harvest to international standards and export directly from the farm instead of a processor.
- To increase the yield of harvest by improving the sorting process to yield a better price for the harvest.
- The distributor/processor can automate the sizing and shelling of the pecan nut.
- To contribute to the agriculture and machine learning community by generating a database of pecan nut images.

1.8. Thesis outline

The remainder of the thesis is arranged as follows:

Chapter 2: Related work: This chapter reviews existing literature in order to build an understanding of how machine learning has been applied in the field of agriculture. It demonstrates what the typical accuracy achieved with Support Vector Machines (SVM), K-Nearest Neighbours (KNN) and artificial neural networks (ANN). The chapter also gives an overview of what type of applications has been considered.

Chapter 3: Research methodology: Describes the methodology approach which was followed. The work done in the different work packages are presented with the software application, which was used.

Chapter 4: A brief introduction to neural networks: This chapter gives an overview of what the differences are between Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning(DL). The chapter also gives the required background and explain the mathematics behind the backpropagation algorithm, which enables a neural network to learn new weights. The chapter serves to as introduction into the next chapter.

Chapter 5: Convolutional Neural network: This chapter extends on the previous chapter but focus on building the understanding of the typical architecture of a convolution neural network (CNN) and how the building blocks like layers, Loss functions and optimisation algorithms work in a modern neural network. The chapter also explains how to use methods like data augmentation to expand a dataset.

Chapter 6: Implementation of a Convolutional Neural Network.: This chapter details the implementation of convolution neural network to classify pecan nut cultivars. The chapter details the hardware and software implementation necessary to complete the research study. It further explains the process of training two different models VGG16 and the MobileNetV2. The chapter concludes with an analysis of the results to answer the main research question and the investigative questions of the research study.

Chapter 7: Conclusions and recommendations: A summary of the findings of this research. A discussion of recommendations for future work is also provided.

Chapter 2: Related work

In this chapter, a brief background is presented on commercial sorting machines to understand the different parts of such a system. An overview of related work is presented, including the different process in a typical automatic sorting algorithm.

2.1. Overview of commercial optical sorting machines

The following section will give some background into the working of a commercial sorting machine.

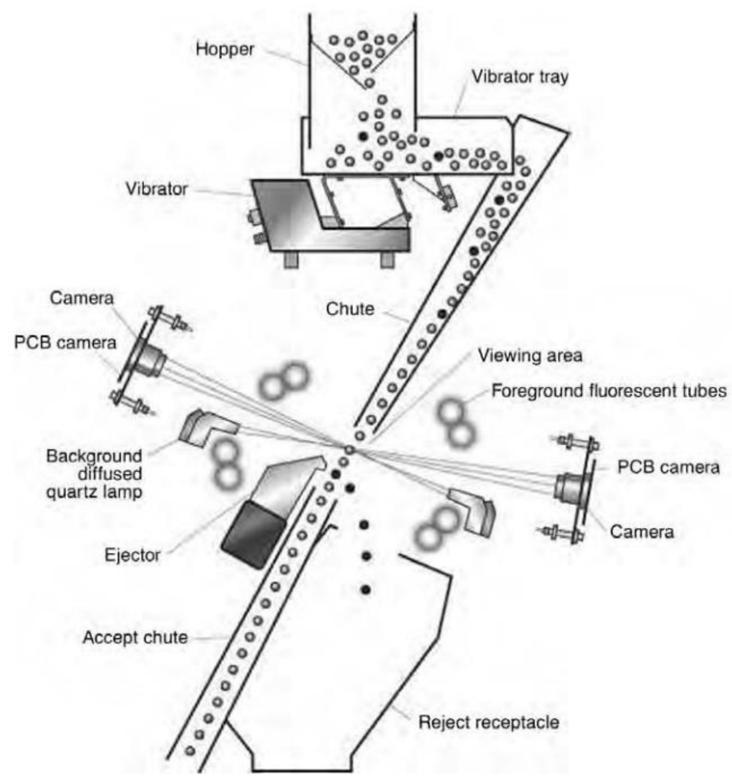


Figure 2-1 Components and layout of a typical sorting machine(Guggisberg and Bosset, 2003, p. 116)

The components and layout of such a system are depicted in Figure 2-1 and grouped below :

- Feed Systems (Hopper, Vibrator Tray, Vibrator and Chute)
- Optics (Camera, PCB Camera, Foreground and background lights)
- Ejection Process (Ejector, Reject receptacle)
- Image Processing algorithms (Not shown)

The following sections will briefly explain how each of these components works and interface with each other.

2.1.1. Feed systems

Dry product (rice, coffee, nuts) are fed into a flat or channelled gravity chute using a vibration hopper. An accelerating belt is used to prevent excessive clumping. Both methods separate the product into a uniform curtain, and this ensures the product is presented at a constant velocity to the optical system(Guggisberg and Bosset, 2003).

2.1.2. Optics

The lenses, lamps and detectors are housed within an optical box to prevent contamination of the optical system. The objects under inspection travel either through or past the optical box. Early optical-sorting machines viewed the product from one side only, which prevented them from detecting defects from the one side(Guggisberg and Bosset, 2003). Modern systems make use of two or three cameras from different angles as the product leaves the chute. This addition increases the accuracy at which the system can identify defects.

2.1.3. Ejection process

To physically remove the unwanted product from the main acceptance stream, short burst of compressed air is emitted through air nozzles aimed directly at the rejects, and they are deflected while in free fall to a reject container(Guggisberg and Bosset, 2003).

2.1.4. Image processing algorithms

In traditional image processing systems, the product either classify as accepted or rejected based on a criterion for colour, or both colour and shape.

The size, cost and complexity of such a system varies depending on the range of particles to be handled, throughput or volume. Typical sorting speeds for something like seeds can be 60kg/hour for a single chute and up to 600kg/hour for a double-chute machine(Guggisberg and Bosset, 2003, p. 118).

The term colour sorting comes from the effect on how the overall product appears. However, the term is misleading. The actual method used is to measure the spectral reflectivity at a particular wavelength, rather than the colour as a whole. The wavelengths cover the visible spectrum (400 to 700nm) and extend into the near infra-red (700 to 1100nm)(Guggisberg and Bosset, 2003). The relative reflectance signal varies from black (zero or no reflectance) to white (100% reflectance).

There are many applications in food sorting, where the defects are similar in colour to a good product. Therefore, more features are required in order to be able to solve different types of applications; one needs to distinguish between size, roundness, area, length.

2.2. Current work in the field

With the improvements in machine learning some manufacturers have started making use of this technology to help the system calibrates itself to account for any irregularities in the environment such as the change product colour over time, calibration drift errors, light source degradation, ambient light or dust accumulation and other real-world processing issues(Toyofuku, Haff and Pearson, 2013, p. 237).

The typical application where optical sorting machines are deployed is to replace human inspectors. For this reason, algorithms used in automated systems are often evaluated based on the system performance as compared to human inspectors. Toyofuka, Haff and Pearson compared the algorithm performance to human sorting for a discriminant analysis-based routine with automatic feature selection(Toyofuku, Haff and Pearson, 2013, p. p237).

Table 2-1 Comparison between automatic sorting and human inspection(Toyofuku, Haff and Pearson, 2013, p. 237)

	False Negatives	False Positives	Overall Error rate
Automatic sorting	19.8%	5.6%	14.4%
Human Inspection	28.3% ±5.7%	2.9% ±2.34%	15.6% ± 2.3%

Table 2-1 are showcasing the variability in performance that is common to human inspectors. The above results indicate comparable or better results and lower variability with automatic sorting than human inspection. As seen in the research and Table 2-1 , human inspectors have a higher chance of not correctly classifying a product with a higher false-negative result where the automatic sorting algorithm tends to be over-optimistic with a higher false-positive result.

The average accuracy achieved by using automatic sorting techniques is 85.6% where humans are slightly lower at 84.4%.

Bhargave, Bansal and Pandey, Naik and Marfatia has conducted thorough reviews of different machine learning methods applied to the fruit and vegetable sector. The reviews looked at each segment of the image processing algorithm and listed the most popular methods with the accuracy achieved for each step(Bhargava and Bansal, 2018), (Pandey, Naik and Marfatia, 2013). The process which was used are shown in Figure 2-2.

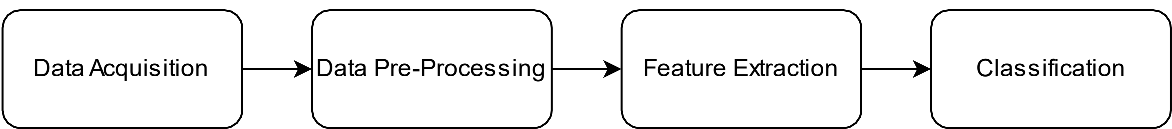


Figure 2-2 Process of fruit and vegetable classification

2.2.1. Data acquisition

The first step in this process is to capture the required data. In food applications, various technologies are used such as camera (CCD and CMOS), ultrasound, magnetic resonance imaging (MRI), electrical tomography and computed tomography (CT) (Bhargava and Bansal, 2018, p. 2).

2.2.2. Data pre-processing

After data has been captured, the acquired images first need to be corrected for distortions and colour. Various filters to reduce noise and median filter, which removes peaks are used. The next step is to segment the images into distinct areas. The primary function is to separate the background from the area of interest. One of the popular segmentation techniques is thresholding and clustering. The performance of fruit images was evaluated by four segmentation methods, as seen in Figure 2-3(Bhargava and Bansal, 2018).

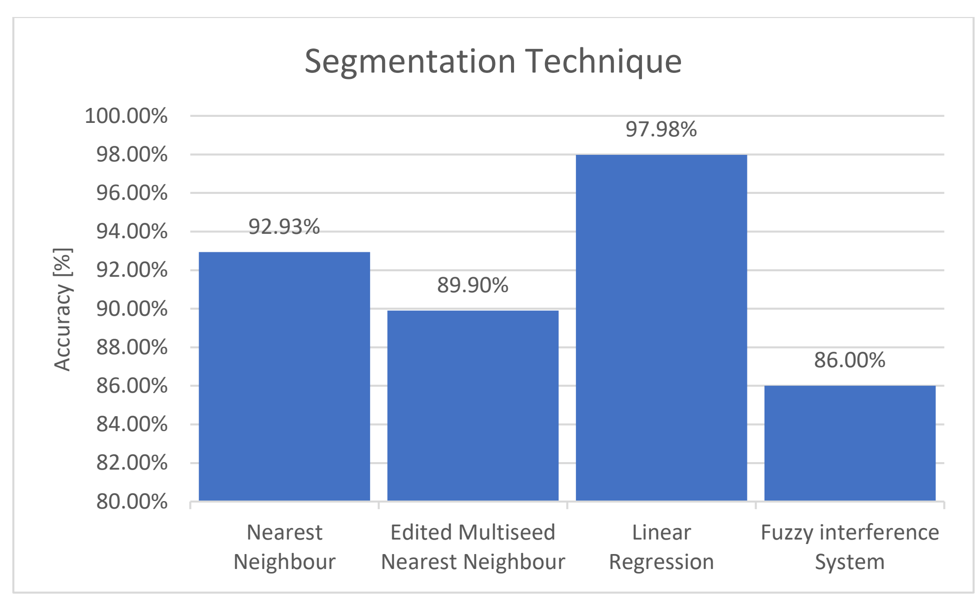


Figure 2-3 Fruit images segmentation techniques (Bhargava and Bansal, 2018, p. 5)

2.2.3. Feature extraction

Certain features are the basics of a computer vision system, as they consist of useful data for image perception, interpretation and object classification. In the food industry colour, textural and morphological (size and shape) are frequently used to analyse the defect and maturity of the fruit and vegetables(Bhargava and Bansal, 2018).

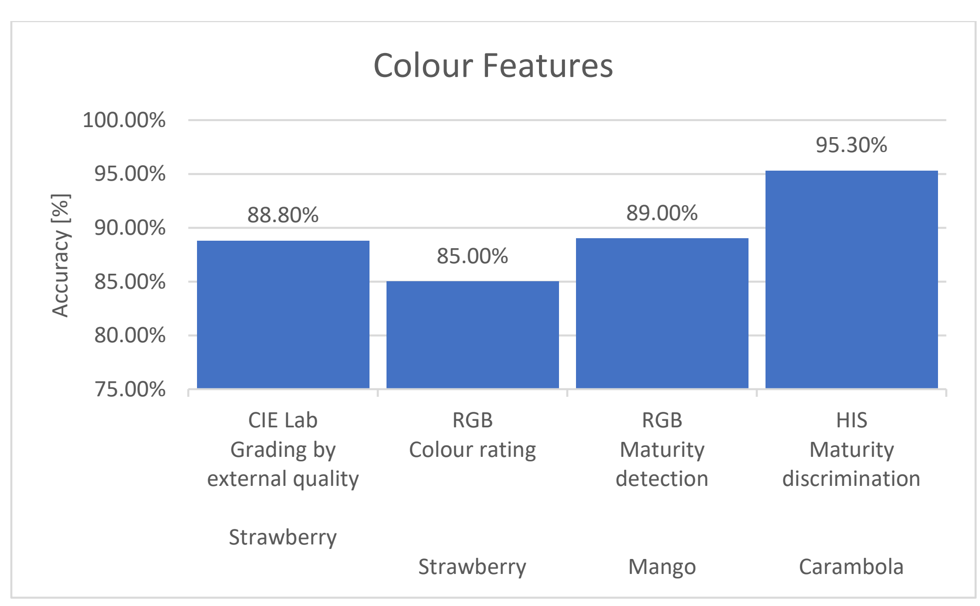


Figure 2-4 Efficiency for quality analysis of fruits and vegetables based on colour features. (Bhargava and Bansal, 2018, p. 6)

From Figure 2-4, the observation could be made that particular colour space is more efficient for a specific fruit or vegetable. As seen, RGB images are used to determine the quality of Strawberries, Mango and Banana, but with varying results. Part of the classification algorithm is to determine what type of colour space should be used for the problem at hand.

2.2.4. Classification

The final step in the process is to classify a product as accepted or rejected based on the set of features. In computer vision, a wide variety of methods: KNN, SVM, Artificial neural networks (ANN) or Convolutional Neural Network (CNN) have been developed for classification in food quality evaluation(Bhargava and Bansal, 2018).

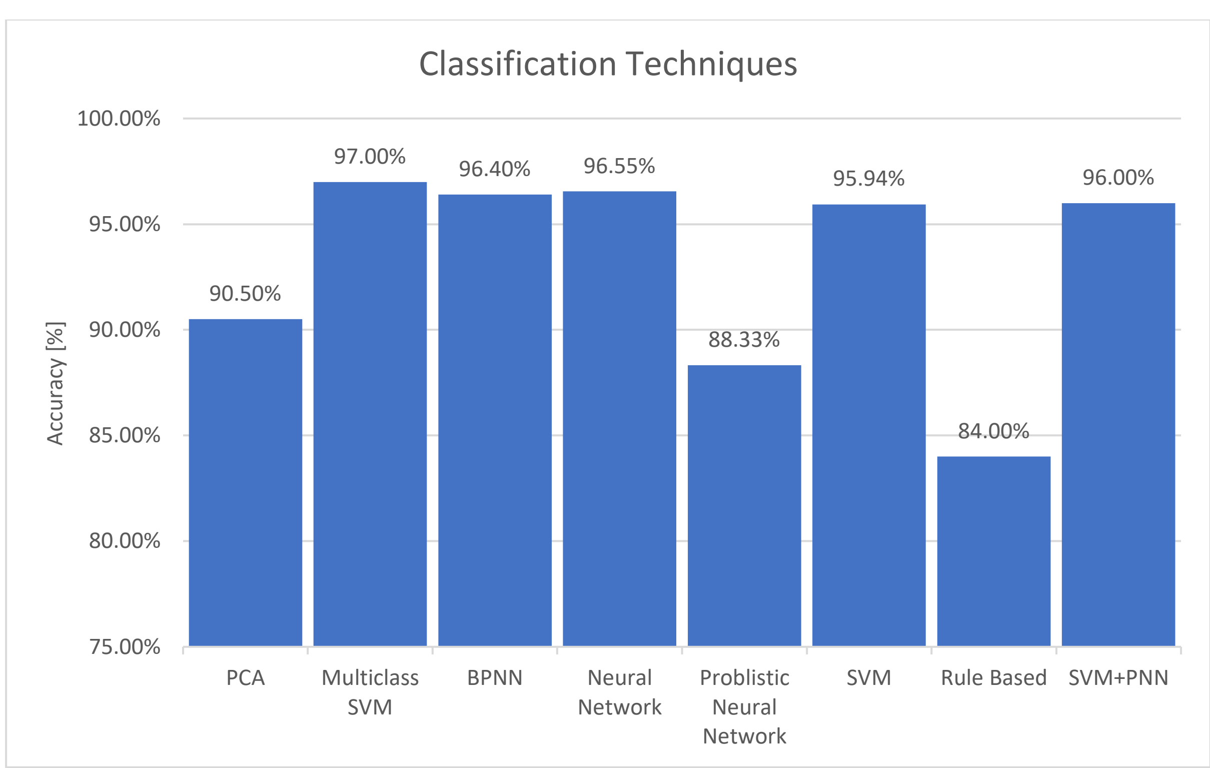


Figure 2-5 Efficiency for quality analysis of fruits and vegetables based on classification techniques (Bhargava and Bansal, 2018, p. 11)

2.3. Work-related to pecan nuts

Mathanker et al. Investigate the use of machine learning classifiers (Adaboost and support vector machine (SVM)) to detect defects in-shell pecan nuts. X-ray images of good and defective pecans, 100 each were segmented, and features were extracted. The linear SVM classifier with the Twice Otsu method gave a slightly better accuracy of 92.7% versus the Real AdaBoost classifier with 92.2%(Mathanker *et al.*, 2011).

Both these classifiers could be suitable for real-time applications as the SVM algorithm needed $10^{-5}s$ and Adaboost only $10^{-6}s$ to classify a defect. Although Mathanker et al. improved the defect detection using X-ray images, their study only focussed on defect and not pecan nut cultivars(Mathanker *et al.*, 2011). Kotwaliwale, Weckler and Brisewitz investigated if x-ray images used as a suitable method for nondestructive quality evaluation of whole pecans(Kotwaliwale, Weckler and Brusewitz, 2006).

To the author's knowledge, no other research could be found relating to using machine vision to distinguish between different pecan cultivars. There are also currently no work using deep learning methods such as Convolutional neural networks to classifies pecan nut cultivars.

2.1. Summary

In this section, an overview was given how a typical commercial sorting machine works and what it consists out. An overview was given of what research is currently being done in the field of agriculture and machine learning. According to the author, there are no other studies which use machine learning to classify pecan nut cultivars, which makes this research project a novel study.

Chapter 3:Research methodology

The following section presents the methodology followed to complete the research objectives. Figure 3-1 shows an overview of all the work packages completed in the project.

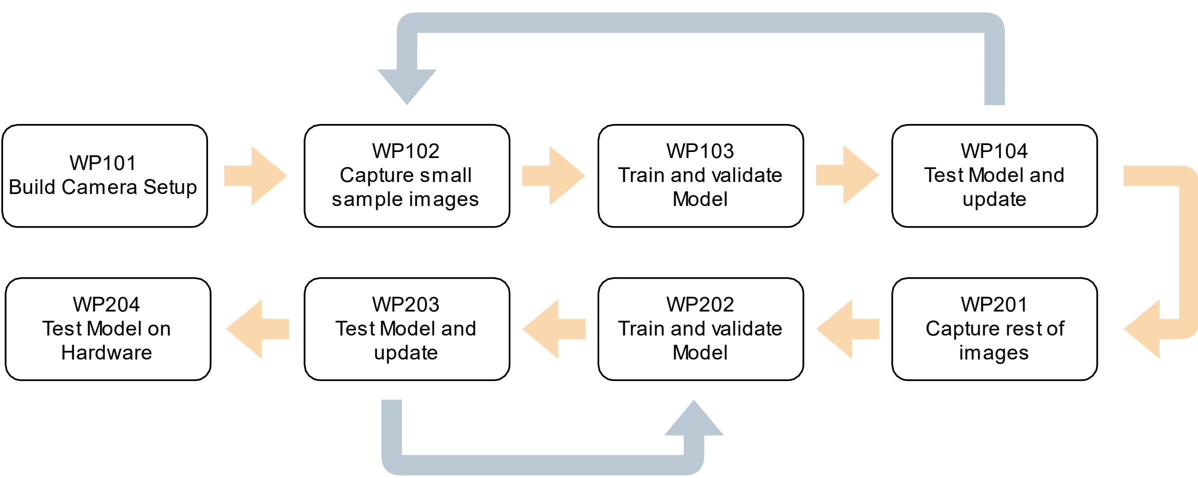


Figure 3-1 Work package breakdown

3.1.1. WP101,WP201 Capturing images

The basis of every machine learning project is useful data, and the bigger the dataset, the better.

A typical machine learning problem can be classified in the following categories :

- binary classification: Output contains only two exclusive classes.
- Multiclass classification: Output can contain more than two classes.
- Scalar regression: output is a prediction of future value in one dimension. E.g
- Vector regression: output is a prediction of future value in more than one dimension.
- Multilabel classification: output can contain more than one class.

As the research project aim is to classify three different pecan nut cultivars, the problem definition falls into the Multiclass classification category.

Before data can be captured and a model trained, one needs to understand the different types of machine learning algorithms, as shown in Figure 3-2:

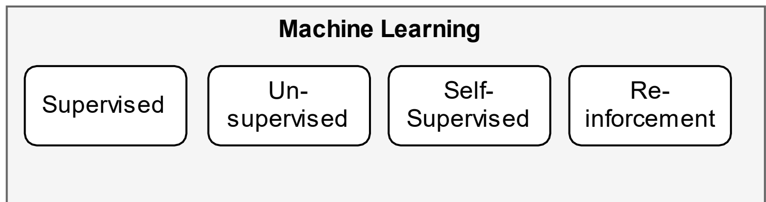


Figure 3-2 Types of ML algorithms

The following types of machine learning algorithms differ in the way the data is labelled, how the neural network will be trained, and what the model be used for.

- Supervised Learning:

Current the most practised and evolved machine learning domain currently. The input is mapped to known outputs by supplying the network with annotated examples, and humans often do the annotation. Applications like image classification and speech recognition are great examples of supervised learning in action. With supervised learning, all training, validation and testing images should be correctly labelled with the specific classes definition, binary, multiclass, scalar for example.

- Unsupervised Learning:

With unsupervised learning, the data is not annotated; hence the network tries and makes sense of the input data without human assistance.

- Self-supervised Learning:

A combination of Super and Unsupervised learning whereby the data is still annotated, but not by human intervention but from the network itself using a heuristic algorithm.

- Reinforcement Learning:

With this algorithm, an “agent receives information about its environment and learns to choose actions that will maximise some reward” (Collet, 2018). The less well-developed field of all the algorithms, however, recent attention from the Google Deepmind project where a model successfully taught itself to play the famous Atari video game, which makes the research into this field exploratory but exciting.

As the research project aims to use visual images of pecan nuts which are of a known cultivar, the type of algorithm used is Supervised learning.

As this is a novel study into pecan nut cultivar classification, there is no database for pecan nuts and a dataset need to be generated. A camera setup was developed consisting of COTS hardware and raspberry PI and a PI-CAM. To better determine what type of features are necessary to capture and at what angles, a small set of images were captured to understand the requirements better.

From experimentation, it was determined that the angle between the horizontal axis of the pecan nut and the camera should not be too shallow, as the visible surface area is increased when the angle is higher. For classification purposes, the more prominent the surface area is the more features such as unique markings there are. The background of the enclosure was chosen as black to minimise the shadow of the pecan nut caused by the light. The LED strip light was also placed uniform around the centre of the enclosure to create an even light from all angles and minimise shadows.

Once a procedure has been established a total of ±990 images, each of 3 different pecan cultivars were taken and labelled.

The images were split into the following batches to maximise the available data available:

- Training set (60%)
- Cross-Validation set (30%)
- Test set (10%)

Whereby the training set is used to complete the first order training, the trained parameters are verified against a cross-validation set where one can start to improve the feature selection of the algorithm without contaminating the training data. The cross-validation set is also used

to prevent overfitting of the training data. Once a suitable accuracy has been achieved, the test set is used to determine the final accuracy of the neural net.

3.1.2. WP103 and WP104 Implement and test model

A significant amount of time was spent acquiring the background knowledge to create and implement a CNN to classify the images.

The following courses were completed to gain the necessary knowledge:

- Machine Learning from Standford University
- Intro into Tensorflow from Google Cloud
- Improving Deep Neural Networks from DeepLearning.AI
- Convolutional Neural Networks in Tensorflow from DeepLearning.AI
- Introduction to TensorFlow for Artificial Intelligence, Machine Learning, and Deep Learning from DeepLearning.AI

The following books were studied.

- Neural Networks and Deep Learning from Michael Nielsen
- Deep learning with Python from Francois Chollet
- Deep Learning for Computer vision with Python 2/3 part series from Adrian Rosebrock

The research was concluded by identifying what nut features will be suitable for this application. Criteria were also needed to be specified how to classify the cultivar and size pecan nuts.

According to the Pecan Breeding & Genetics, Agricultural Service, U.S. Dept of Agriculture, the criteria to determine a pecan nut cultivar with the dimension is a below.:

Descriptors for the pecan nut shape based on nut length to height ratios.

- Orbicular 1 to 1.39
- Ovate 1.40 to 1.59, widest at base
- Obovate 1.40 to 1.59, widest at the apex
- Oval elliptic 1.40 to 1.59, widest in middle
- Elliptic 1.60 to 1.79
- Oblong elliptic1.80 to 1.99

- Oblong greater than 2.00

Descriptors of apex and base shape are very rudimentary;

- "acute" for angles sharper than 90 degrees
- "acuminate" for acute angles having concave surfaces; and
- "obtuse" for angles greater than 90 degrees.

Cross-section form is described as:

- "round" if nut height to width ratios are between .95 and 1.10,
- "laterally compressed" if nut height to width ratios exceed 1.10, and as
- "flattened" if they are .95 or less.

The definition for each of the cultivars in the research study is:

- Mahan: oblong, with acute apex and base; nut often asymmetric, appearing 'pinched' in the middle due to flattening of abaxial and adaxial surfaces; flattened in cross-section
- Shoshoni: oval elliptic with obtuse apex and rounded base; laterally compressed in cross-section
- Wichita: oblong, with acute to acuminate, asymmetric apex and rounded apiculate base; round in cross-section

The implementation of the Convolution neural network made use of the following software libraries in the Python programming language.

- OpenCV :

"OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products." (https://opencv.org/about/).

- Numpy :

"NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays

and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.” (https://numpy.org/devdocs/user/whatisnumpy.html, accessed 2020-04-12)

- Tensorflow:

Tensorflow is an open-source, high-performance library for numerical computation. These qualities make the library highly applicable to machine learning problems.

Figure 3-3 shows the software stack used for this research project. This project made use of a model-level library called Keras. Keras provides high-level building blocks for developing deep-learning models. The library can interface to various lower-level numerical computation libraries like TensorFlow from Google, Theano from the MILA lab at Universiè de Montreal or the Cognitive Toolkit(CNTK) developed by Microsoft. The models developed in Keras are able to run via TensorFlow (or Theano, or CNTK) on GPUs or CPUs. On GPUs Tensorflow interface to a well-optimised deep-learning library developed by NVIDIA called CUDA Deep Neural Network Library (cuDNN).

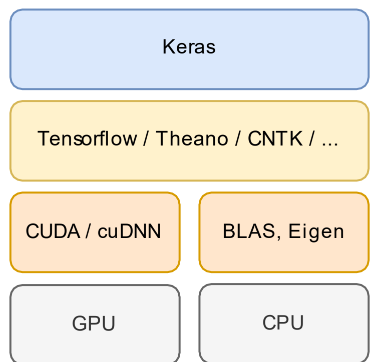


Figure 3-3 Software and hardware stack (Collet, 2018, p62)

For the research project, Tensorflow was used with an extension to Keras, and the computer used the CUDA and cuDNN libraries which utilised the Nvidia Geforce GTX1050ti GPU.

The neural network is tested with the cross-validation data set, and improvements to the pre-processing and hyperparameter selection are made to improve the performance. As can be seen in Figure 3-1, the training of a neural net is an iterative process of testing, updating and validating.

3.1.3. WP202 and WP203 Improve model

After all, the pecan nuts were recorded, and a dataset was created, the next focus was to improve the model. This in itself is an iterative process of testing, updating parameter selection and validating.

3.1.4. WP204 Validate model on hardware

Once a suitable accuracy was achieved, the model was validated on the actual hardware in real-time.

3.1.5. WP401 and WP402

Up to this stage, all training, validating and testing had been performed on a personal computer. The next step is to move to the hardware solution and verify the neural net performance in a real-time scenario.

3.2. Summary

In this section, A methodological approach was presented to how the research were completed and what each step entailed.

The next section will give the necessary background to understand how neural networks are able to learn new features during the training process.

Chapter 4: A brief introduction to neural networks

To understand what solution were implement in the project, one needs to understand what the difference is between Artificial intelligence, machine learning and deep learning.

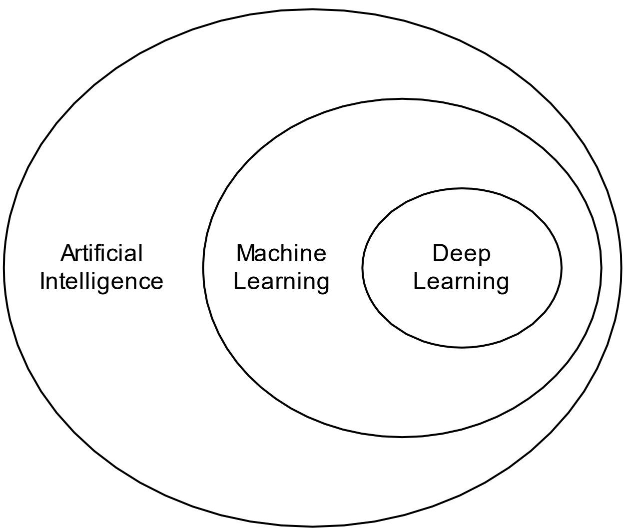


Figure 4-1 Overview AI, Machine and Deep Learning inspired by (Collet, 2018, p4)

Deep learning is a small section of a larger field called machine learning which belongs to a more significant field called artificial intelligence, the Venn diagram in Figure 4-1 shows the relationship between these fields.

A concise definition of the artificial intelligence field would be " the effort to automate intellectual tasks normally performed by humans" (Collet, 2018). Although the field encompass machine learning the scope includes approaches which do not involve any learning. Early chess programs which made use of explicit rules were not classified as machine learning but contained intelligence which mimics human actions. From 1950 to 1980, experts such as Newell and Simon believed that human-level artificial intelligence could be created by defining a sufficiently large enough set of explicit rules for manipulating knowledge(Newell and Simon, 2007). The approached were known as symbolic AI. Although symbolic AI worked well for well-defined problems as playing chess, this approach deemed not suitable for solving more complex problems like image classification.

Francois Chollet gives a great explanation of the difference between symbolic AI and machine learning as seen in Figure 4-2. With symbolic AI human's input the rules and data and the

output are the answers. However, with machine learning the humans input the data and answers and outcome the rules(Collet, 2018,p5).

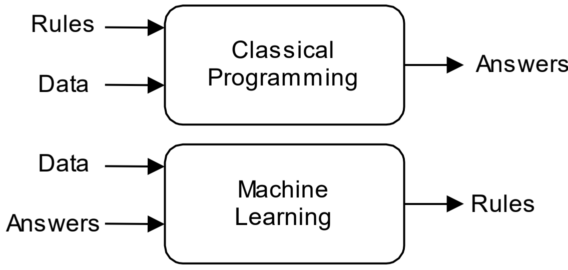


Figure 4-2 Machine Learning a new paradigm (Collet, 2018, p5)

One of the classes of machine algorithm is Artificial Neural Networks (ANNs) which learns from data and specialises in pattern recognition. The structure and function of the neural network were inspired by the working of the human brain Figure 4-3. The figure illustrates the similarities between a Neuron in an ANN and the human brain.

The neuron in the human brain is called the soma, and each soma has inputs (dendrites) and outputs called axons. The inputs and outputs connect the soma to other somas in the brain. If the neuron receives electrical input from a dendrite which is sufficiently powerful enough to activate the neuron, the neuron will pass the signal on to other neurons via an axon. These binary activations inspired the working of a neuron in a neural network.

It is important to note that as useful as this illustration is, the human brain is far more complicated as this similarity portrays.

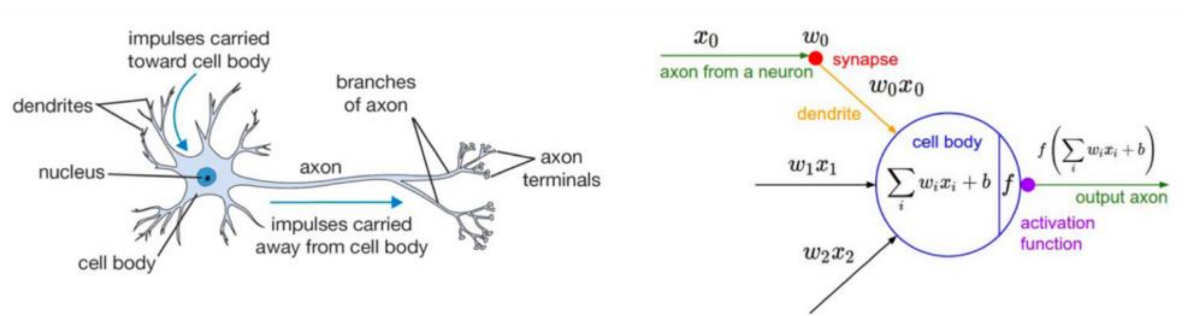


Figure 4-3 Biological Neuron with the inspired mathematical model

McCulloch and Pits presented in 1943, what is considered the first neural network model. The model was capable of classifying or recognising two different categories from some input. Although groundbreaking at that stage, the model required a human to adjust the parameters(weights) to classify a specific input category correctly (Warren S. McCulloch and Walter Pitts, 1943).

Rosenblatt solved the problem with his Perceptron neuron in 1958, where his model would correctly classify an input by automatically learning the parameters(weights) without any human intervention (F. Rosenblatt, 1958).

An example of the architecture is shown in Figure 4-4.

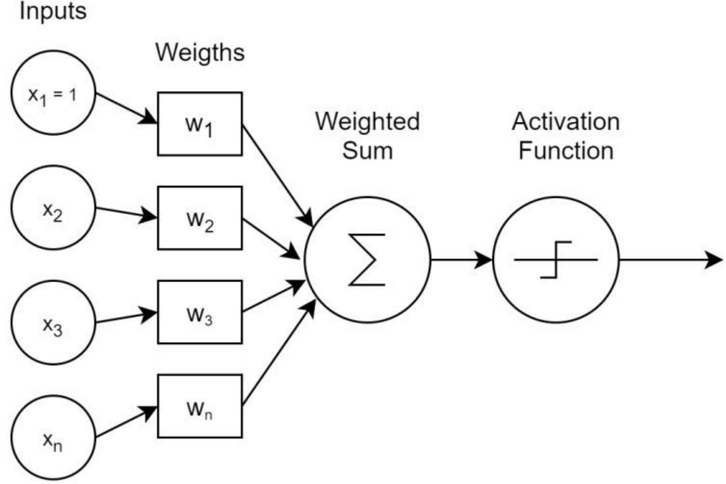


Figure 4-4 A Perceptron neuron.

The Perceptron consists out of the following elements see Figure 4-4, Input Nodes $\{X_1,\ldots, X_n\}$. Weights $\{W_1,.., W_n\}$, Weighted Sum and an activation function as seen in Figure 4-4. The input nodes are multiplied with their respective parameters called weights and then added together in the summation block. The activation function in Rosenblatt case was a step function σ which produced a binary output as seen in eq 1 and 2 (Nielsen, 2015).

$$S = \sum_{i=1}^{n} x_i \, w_i \tag{1}$$

$$\sigma(s) = \begin{cases} 1 \text{ } if \text{ } S \geq 0 \\ 0 \text{ } if \text{ } S < 0 \end{cases} \quad (2)$$

The principle that the activation function either activates, based on a set of inputs and weights or not is where the resemblance of a biological neuron was made. The Perceptron, as seen in Figure 4-4, is called a neuron. This neuron could be assembled in different patterns; one of these patterns is to connect multiple neurons in different layers together, hence an artificial network.

In 1969 Minsky and Paper published a paper which identified a crucial problem with the Perceptron algorithm. Although the algorithm can learn new parameters(weights), it is unable to solve non-linear problems. This drawback makes the Perceptron algorithm unsuitable for image classification, as the image classification is inherently a non-linear problem (Marvin Minsky and Seymour Papert, 1970).

Individual research from Werbos (P. J. Werbos, 1974), Rumelhart (Rumelhart, Hinton and Williams, 1986), and LeCun (Yann LeCun et al., 1996) were able to solve this issue with their research in the backpropagation algorithm enabled multi-layer-feedforward neural network and by making use of non-linear activation functions. An example of such a network is shown in Figure 4-5.

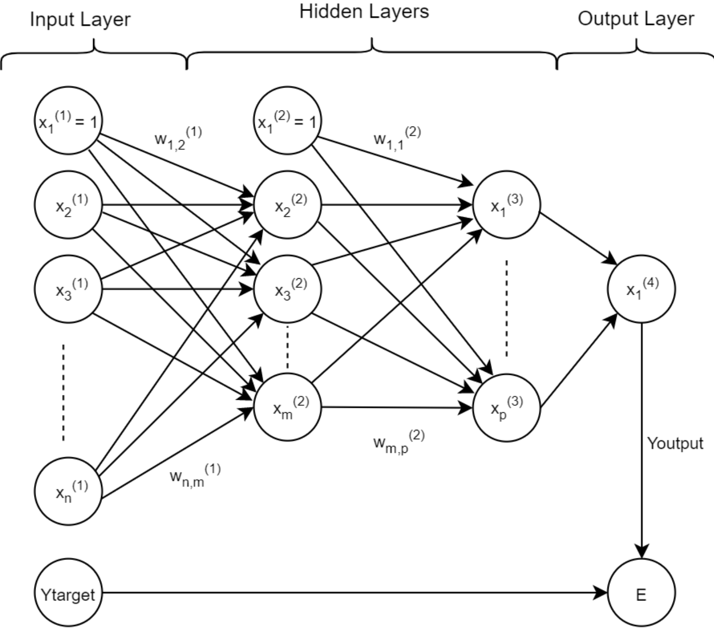


Figure 4-5 Multilayer feedforward Network

The multilayer feedforward network or artificial neural network (ANN) consists of multiple perceptron neurons; in this configuration, they are called nodes. These nodes are stacked sequentially in layers, and connections are made between the nodes.

The backpropagation algorithm is the basis of modern-day neural networks, which allows us to train the parameters(weights) required for accurate image classification efficiently. The complete working of the algorithm is explained in section 4.2.

LeCun (Yann LeCun et al., 1996) laid the foundation with his research in Convolution Neural Network, where he successfully applied It to recognise handwritten characters. His network was able to automatically learn discriminating patterns called “filters” from images by stacking layers on top of each other. Filters in the lower layers would extract edges while higher-level layers used the edges to learn more abstract features.

Two building blocks in any neural network is the activation functions and the backpropagation algorithm; the rest of this chapter is devoted to describing these functions in details. The other building blocks, such as layers and optimisers, are described in chapter 5 with the inner working of a convolution neural network (CNN).

4.1. Activation functions :

One of the main reasons why ANNs can achieve such high accuracy is work done on developing different activation functions. In modern-day neural networks, there are a few activations functions in use. The choice of activation function depends on what type of classification problem one is trying to solve. The following are two examples of activation functions commonly found in image classification problems :

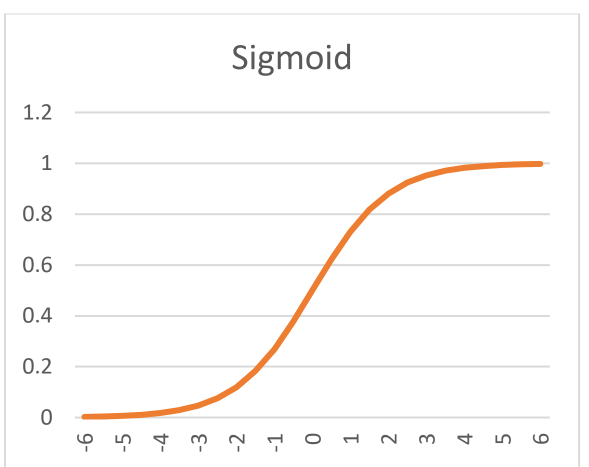


Figure 4-6 Sigmoid activation function

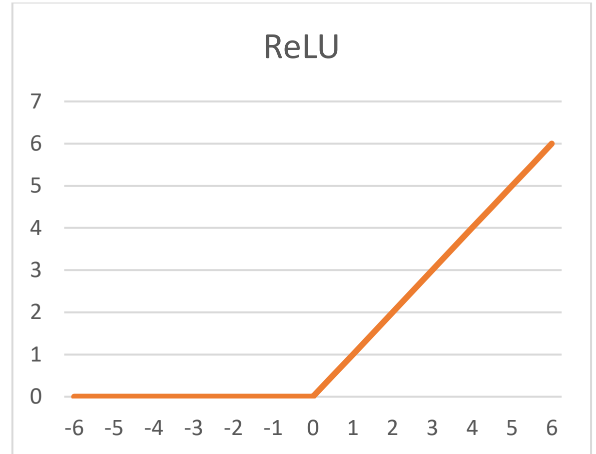


Figure 4-7 ReLU activation function

- The sigmoid activation function Figure 4-6 and eq 3 has two advantages for learning above the step function. The function is continuous, differentiable and asymptotically approaches its saturation values. The sigmoid function also has two significant disadvantages such the outputs are not zero centred, which slows down optimisation as the gradient goes either positive or negative. Furthermore, if the output neuron saturates the gradient becomes virtually zero. This phenomenon is called diminishing of gradients which causes the learning process to stall.

$$\sigma(s) = \frac{1}{1 + e^{-s}} \tag{3}$$

- The ReLU (eq 4) also known as “ramp functions” as can be seen in Figure 4-7. The output of the function is zero for negative inputs and linearly increases for positive values. Because of this behaviour, the function removes all negative information which makes it unsuitable for all types of datasets. The function non-saturating form prevents the gradient not to vanish or explode when used in backpropagation.

$$\sigma(s) = \max(0, s) \tag{4}$$

4.2. Backprogration algorithm

The backpropagation algorithm enables the neural network to learn by propagating the error between the predicted output and actual output back throughout the network.

The process consists of two steps a forward pass (section 4.2.1) where the input is propagated through the network to predict an output, and a backward pass(section 4.2.2) where the error between the predicted output and actual output is used to update the weights and biases of each node.

Both these steps are described in the next section for better understanding.

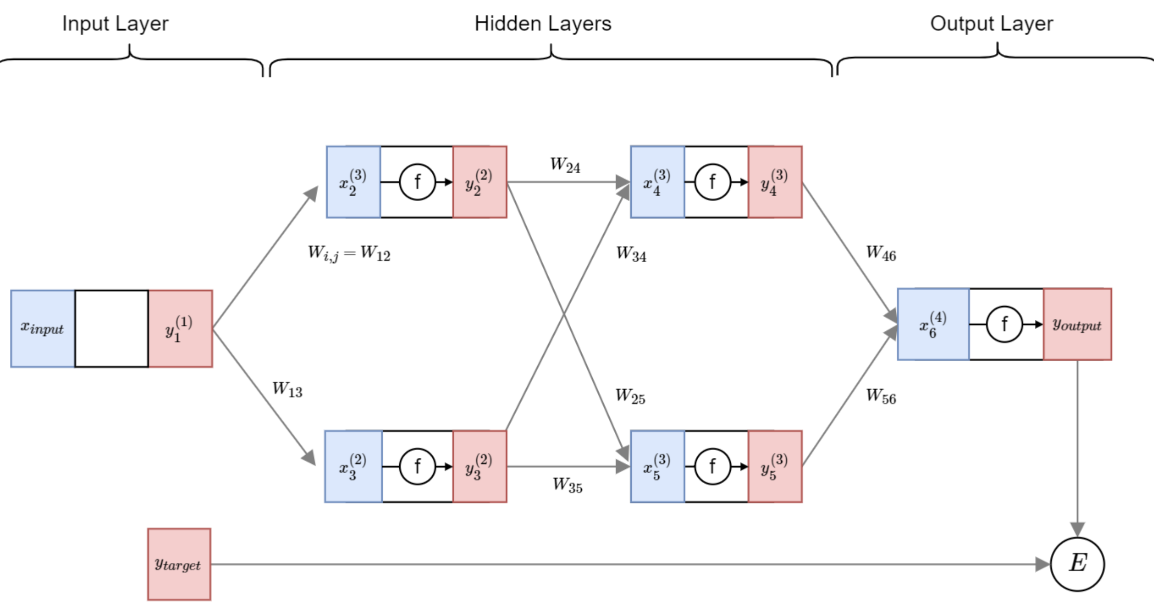


Figure 4-8 Forward Pass

Figure 4-8 shows a neural network with one input, two hidden layers with two nodes each and an output layer with a single node. Note the symbol $x_i^{(L)}$ and $y_i^{(L)}$, in the figure represents the i-th node in the L-th layer. The connections between the layers are called weights, shown as $W_{i,j}$, the connections are made from a i-th node in the L-th layer to j-th node in the $(L + 1)$-th layer. Because of the forward connections between the layers the network is also called a feedforward network. Each node (perceptron neuron) has multiple input values of x, an activation function $f(x)$ and an output $y = f(x)$. To enable the neural network to learn sophisticated features as needed in image classification the activation function $f(x)$ should be an non-linear function as mention before in section 4.1, as the sigmoid function eq 3.

The objective is to learn the weights of the network automatically by minimising error (E) between the predicted output y_{output} and the y_{target} for all inputs x_{input}.

4.2.1. Forward pass:

The process begins with the forward propagation step where the x_{input} is taken as an input to the neural network. The input node is like any other node, but without an activation function, the output is then equal to the input, i.e $y_1^{(1)} = x_{input}$. The first node in the hidden layer is updated by taken the output of the previous layer and the weights to compute the input x of the node with eq 5.

$$x_i^{(L)} = \sum_{j=1}^{C} x_j^{(L-1)} W_{j,i}^{(L-1)} + b_j^{(L)} \tag{5}$$

Where C is the total number of nodes in a layer $(L-1)$ connected to node $x_i^{(L)}$ .The output of the hidden layer node is updated by:

$$y = f(x) \tag{6}$$

Eq 5 multiplies each output and weight of the previous layer adds a bias value, which for the example is set to 1. Then iterates through all the input connections to that node and adds the outputs together, into a single value $x_i^{(L)}$.

By making use of EQs 5,6, the output of each node is propagated through the rest of the network until the final predicted output of the network is calculated.

4.2.2. Backward pass:

The backward pass calculates the error between the Predicated output and actual output. The difference is used to update the weights and biases of each node. This process is shown in Figure 4-9.

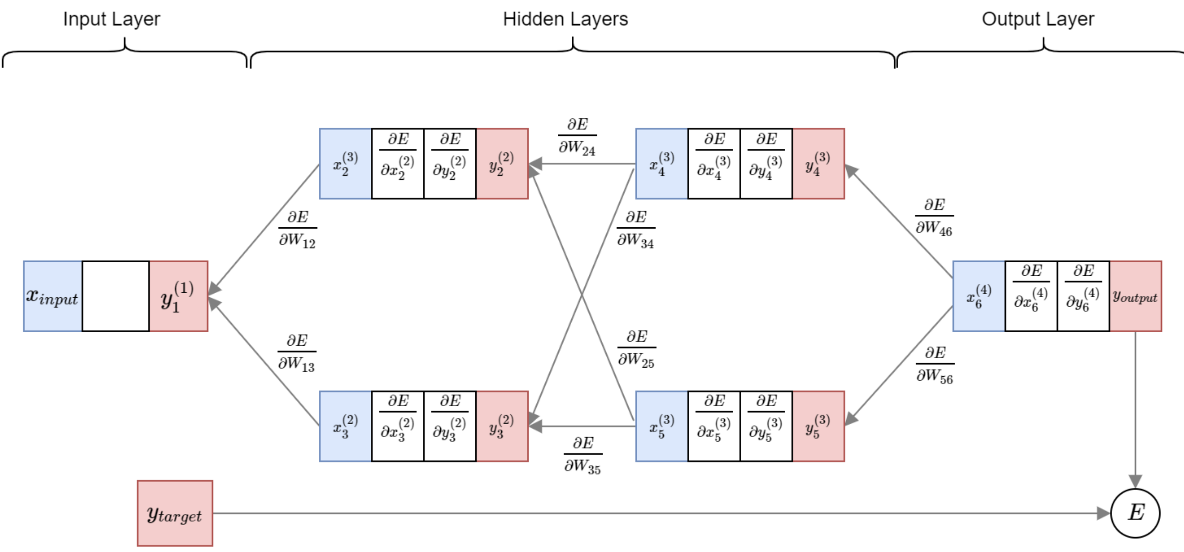


Figure 4-9 Backward Pass

The first step is to calculate the error between the predicted and the actual output with eq 7:

$$E(y_{output}, y_{target}) = \frac{1}{2}(y_{output} - y_{target})^2 \tag{7}$$

The next step is to find out how the error change with the predicted output with eq 8:

$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target} \tag{8}$$

The backpropagation algorithm is used to calculate how much each weight contributed to the overall error by taking the partial derivative of the error with respect to each weight. The derivatives for each node is calculated from the output layer back to the input layer, hence backward pass.

To help compute $\frac{\partial E}{\partial W_{i,j}^{(L)}}$ for each node, two additional values are stored as seen in Figure 4-9. The two values are for how much the error changes with :

- the total input of the node $\frac{\partial E}{\partial x_i^{(L)}}$
- the output of the node $\frac{\partial E}{\partial y_i^{(L)}}$

The next step is to use the chain rule to calculate how the output of each node change with the input of the node with as shown in eq 9 :

$$\frac{\partial E}{\partial x_i^{(L)}} = \frac{\partial y_i^{(L)}}{\partial x_i^{(L)}} \frac{\partial E}{\partial y_i^{(L)}} = \frac{\partial}{\partial x_i^{(L)}} f(x) \frac{\partial E}{\partial y_i^{(L)}} \tag{9}$$

Where the function $f(x)$ in $\frac{\partial}{\partial x_i^{(L)}} f(x) = f(x)(1 - f(x))$ is a sigmoid activation function.

The error derivative with respect to the total input of a node eq 9 is used to calculate the error derivative with respect to the weights coming into that node with eq 10:

$$\frac{\partial E}{\partial W_{i,j}^{(L)}} = \frac{\partial x_i^{(L)}}{\partial W_{i,j}^{(L)}} \frac{\partial E}{\partial x_i^{(L)}} = y_i^{(L)} \frac{\partial E}{\partial x_i^{(L)}} \tag{10}$$

By using the chain rule again, the error derivative with respect to the input of the previous layer can be calculated with eq 11 :

$$\frac{\partial E}{\partial y_i^{(L)}} = \sum_{j=1}^{C} \frac{\partial x_j^{(L)}}{\partial y_i^{(L)}} \frac{\partial E}{\partial x_j^{(L)}} = \sum_{j=1}^{C} W_{i,j}^{(L)} \frac{\partial E}{\partial x_j^{(L)}} \tag{11}$$

Once all the derivatives are calculated, the weights and biases are updated by making use of the gradient descent function shown in eq 12:

$$W_{i,j}^{(L)} = W_{i,j}^{(L)} - \alpha \frac{\partial E}{\partial W_{i,j}^{(L)}} \tag{12}$$

Where α is a positive constant called the learning rate, the value is fined tuned empirically.

Gradient descent is an iterative optimisation algorithm for finding the global minimum of a function. The error gets minimised by taking steps proportional to the negative of the gradient of the function at a certain point. Simply put: if the error (E) goes down when the weight increases ($\frac{\partial E}{\partial W_{i,j}^{(L)}} < 0$), then increase the weights, otherwise decrease the weight. The gradient algorithm is discussed in more detail in section 5.5.

4.3. Summary

In this section, a brief overview of the history of neural networks was discussed, and where machine learning and deep learning fit into the larger artificial intelligence field. The first neural network called a Perceptron were discussed and explained. Although important from a historical perspective, the algorithm had one major disadvantage, the inability to classify non-linear separable points.

For a machine-learning algorithm to handle more complex datasets, two elements are required :

- non-linear activation functions and a
- multi-layer network.

For a neural network to be able to learn the weights automatically, a backpropagation algorithm needs to be implemented, which consists of two phases :

1. The forward pass where the input image is propagated through the network to obtain a predicted output class.
2. A backward pass where the gradient of the error is computed and the weights in individual nodes are updated by using the chain rule and the gradient descent algorithm.

In the next section, a unique kind feed-forward network is discussed called a Convolutional Neural Network (CNN). CNN’s are the industry standard for optical classification problems in the field of deep learning.

Chapter 5: Convolutional Neural network

In the previous section, the traditional feed-forward network was discussed, and a brief background was given. In such a network, all the neurons from the input layer are connected to all output neurons in the next layer, the technical name for this is a fully connected (FC) layer. This type of connection does not work well for images as each pixel would need to be connected to a neuron and, If the input image size is 224 x 224 x 3 (224 pixels wide, 224 pixels with three colour channels), it would mean if the input layer were an FC layer, it would have 224 x 224 x 3 + 1 = 150,529 parameters which need to be trained. To limit the number of connections, i.e. weights a certain kind of layer is used called a convolutional layer to help reduce the number of parameters for each layer.

A convolutional layer can be thought of as a filter, which isolated or enhances certain aspects of the input image and during training, a CNN can automatically learn the specific values for these filters.

A convolutional neural network (CNN) is thus defined as a neural network where at least one of the FC layers are swapped with a convolutional layer. A nonlinear activation function is applied to the output of the convolutions, this process is repeated by stacking a convolution layer + activation function until the end of the network where there are one or more fully connected layers from which the final output classification can be made. Below is an example of an AlexNet-like CNN architecture :

```
INPUT => [CONV => RELU => POOL] *2 => [CONV => RELU] *3 => POOL => [FC => RELU => DO] *2 => SOFTMAX
```

The exact working of each of these layers is described later in this chapter.

In deep learning, a CNN can learn to detect edges in the lower layers, and from the edges to detect high-level features such as eyes, ears, facial elements, for example.

CNN’s has a few inherent benefits above traditional machine vision approaches such as local invariance and compositionality. Local invariance is the ability to detect edges or features irrespective where they are in the image. This reason for this is because all the activations of the pixels in the convolutional layer are pooled together in a pooling layer which is discussed

later. Compositionality does not affect a CNN. The reason is that the convolution operation slide from left to right and top to bottom, where the filter will respond when coming across the edges and corners irrespective where they are in the image.

5.1. Common architectures

The following section will give a quick background and overview of the architectures used in the research study. Two architectures were used called VGG16 and MobilenetV2. Both these architectures are world-class classifiers trained on thousand to millions of images and are able to classify up a thousand different classes (VGG16). However, not one of these architectures has been taught to classify different pecan nut cultivars which are the aim of this research study. Later in the chapter (see section 5.9) the method of transfer learning is described which were used to retrain these classifiers to distinguish between different pecan nut cultivars.

5.1.1. Visual Geometry Group Network (VGGNet)

Karen Simonyan, and Andrew Zisserman from the Visual Geometry Group (VGG) at the University of Oxford published the VGGNet in 2015 after winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014 (Simonyan and Zisserman, 2015). The ImageNet challenge is an extensive database used for research computer vision. Figure 5-1 shows the VGG16 architecture starting from the left-hand side the input image and progressing to the right where a 1x1000 vector holds the probability of the specific class in the input image.

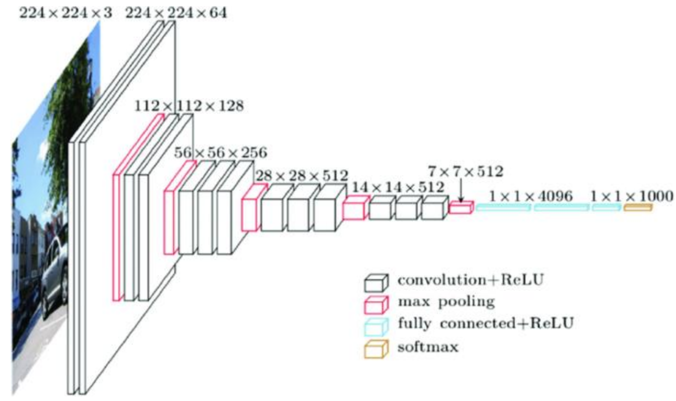


Figure 5-1 VGG16 Architecture (Loukadakis, Cano and O 'boyle, 2018)

The VGGNet was the first network that showed that it is still possible to achieve a high accuracy classification with smaller 3x3 kernels, up to this point all the previous networks like

AlexNet (Krizhevsky, Sutskever and Hinton, 2017) the 2012 winner used 11x11 and 5x5 kernels in the first two layers. Moreover, ZFNet(Zeiler and Fergus, 2013) the 2013 winner of ILSVRC used 7x7 kernels. The smaller size kernels reduced the number of parameters and therefore, the size of the network significantly.

5.1.2. MobilenetV2

A. Howard et al. from Google research. Published in 2017, a lightweight network called MobileNet (Howard *et al.*, 2017). The purpose of the research was to develop a CNN for mobile and embedded vision applications. Mobilenet is a lightweight deep neural network based on a streamlined architecture that used depth-wise separable convolutions. Figure 5-2 shows the architecture of the MobileNetV1 neural network.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
5× Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 5-2 MobilenetV1 Architecture (Howard et al., 2017)

The difference between a standard convolution and depth-wise separable convolution is shown below in Figure 5-3.

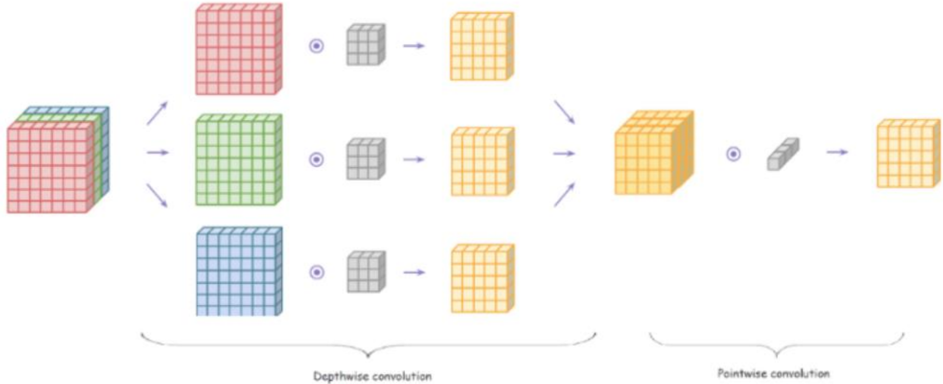


Figure 5-3 Depth-wise Separable Convolution (https://towardsdatascience.com/deep-dive-into-the-computer-vision-world-f35cd7349e16)

With Depth-wise separable convolution operation, the three channels of the input are split. Each channel is then separately convolved with a corresponding filter and then concatenated together. The last step of the process to complete a pointwise convolution. The improvement of this process is computational cost, according to A. Howard et al. the MobileNet architecture require 8-9 times less computation than a network with standard convolutional layers, with a negligible impact on accuracy.

In 2018 M. Sandler et al. published an improved version of the Mobilenet architecture called MobileNetV2 (Sandler *et al.*, 2018). Figure 5-4 shows the MobileNetV2 architecture, where t is the expansion factor, C the number of output channels, n the repeating number and s the stride.

The original network was extended with two new ideas, Inverted Residuals and Linear Bottlenecks layers.

Both those two concepts are advanced and require a significant background to comprehend fully, and the full details are described in the MobilenetV2 paper (Sandler *et al.*, 2018), the background details are considered outside the scope of this research.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 5-4 MobileNetV2 architecture (Sandler et al., 2018)

The next section will explain the building blocks of CNN's namely :

- Layer types
- Loss functions
- Optimisation algorithms

The chapter will conclude to look at how to improve the performance of the neural net given a small dataset with different regularisation approaches. A more in-depth explanation would be given why a CNN is invariant to rotation, scale and translation in the dataset, and what the difference is between machine learning and deep learning regarding the required features in an image. Finally, an overview of different training methods is presented to give the necessary background in the methodology followed for the research project.

However, before the layers are explained, it is necessary to understand what convolutions operations are precise.

5.2. Understanding convolutions:

In the machine vision and image processing field, convolutions are used to filter an image to enhance a specific aspect of the image, e.g. To blur or smooth an image or to detect edges. This effect is realised by completing an element-wise multiplication operation with a kernel (n x n matrix) and a section of an image (m x m matrix) and summing the elements together as shown below.

$$\begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix} * \begin{bmatrix} 107 & 16 & 71 \\ 231 & 47 & 215 \\ 60 & 148 & 2 \end{bmatrix} = \sum \begin{bmatrix} 0 & 16 & 142 \\ 0 & 47 & 430 \\ 0 & 148 & 4 \end{bmatrix} = 787$$

The image is processed by sliding the kernel over the image from top left to the bottom right a pixel at a time, and then repeating the convolution process as shown below.

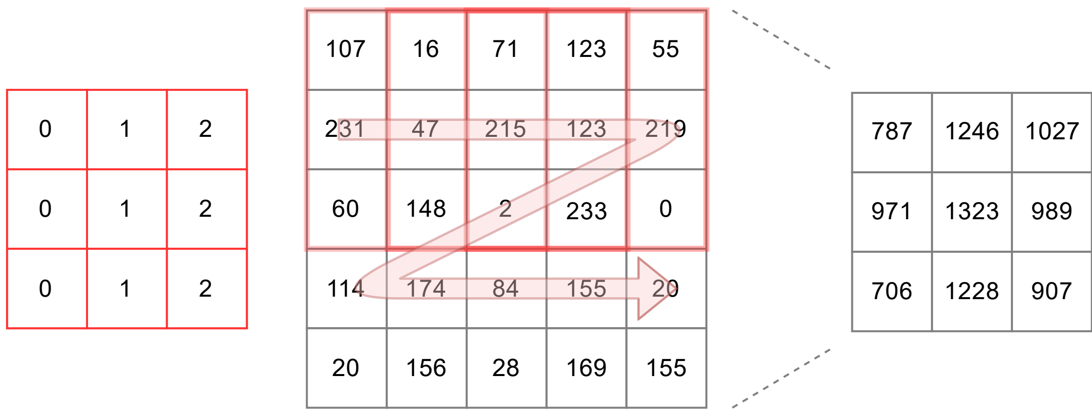


Figure 5-5 Convolve operation no padding LEFT: Kernel, Middle: Original matrix, Right: Output matrix

Sliding the kernel across the image decreases the spatial dimension of the image where the 5x5 input matrix as decreased to a 3x3 output matrix, as seen in Figure 5-5. This effect is helpful to decrease the size of the images in a CNN layer as the number of parameters which need to be trained also decreases. However, the side effect is that it becomes impossible to build and train deep neural networks, as the size of the image becomes too small to learn suitable features. For the layers where the output dimension needs to stay the same as the input image, the concept of padding is introduced.

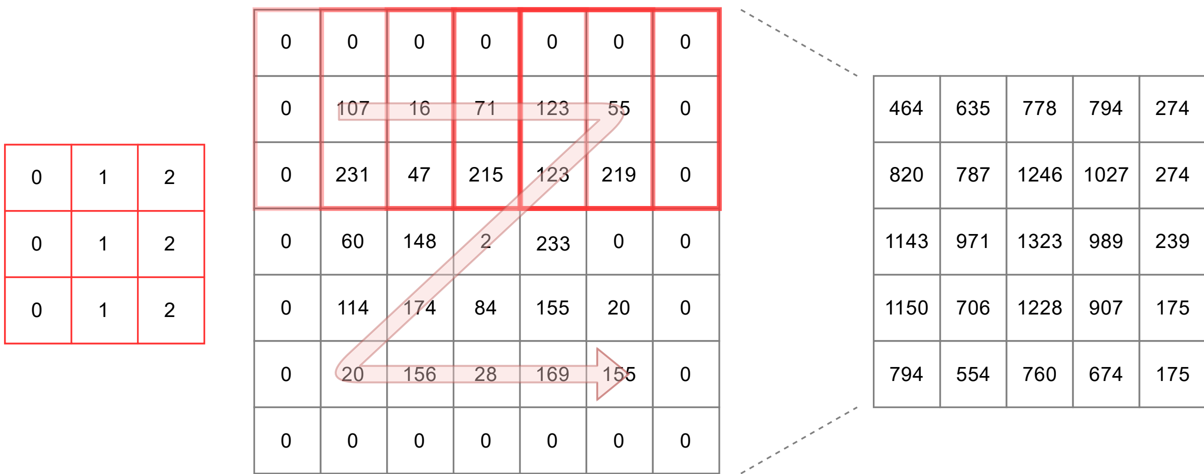


Figure 5-6 Convolve operation zero-padding LEFT: Kernel, Middle: Original matrix with zero padding, Right: Output matrix

There are different types of padding methods like replicate padding, where the outside pixels are replicated on the border of the image. Zero-padding, as seen above, is where a zero value border is applied to the outside of the image, or wrap-around padding where the border pixels are the same as the opposite side pixels. Figure 5-6 shows when the original 5x5 matrix is padded with zeroes increasing the size to a 7x7 matrix, the output matrix size is 5x5 which is the same size as the original matrix before the convolutional operation.

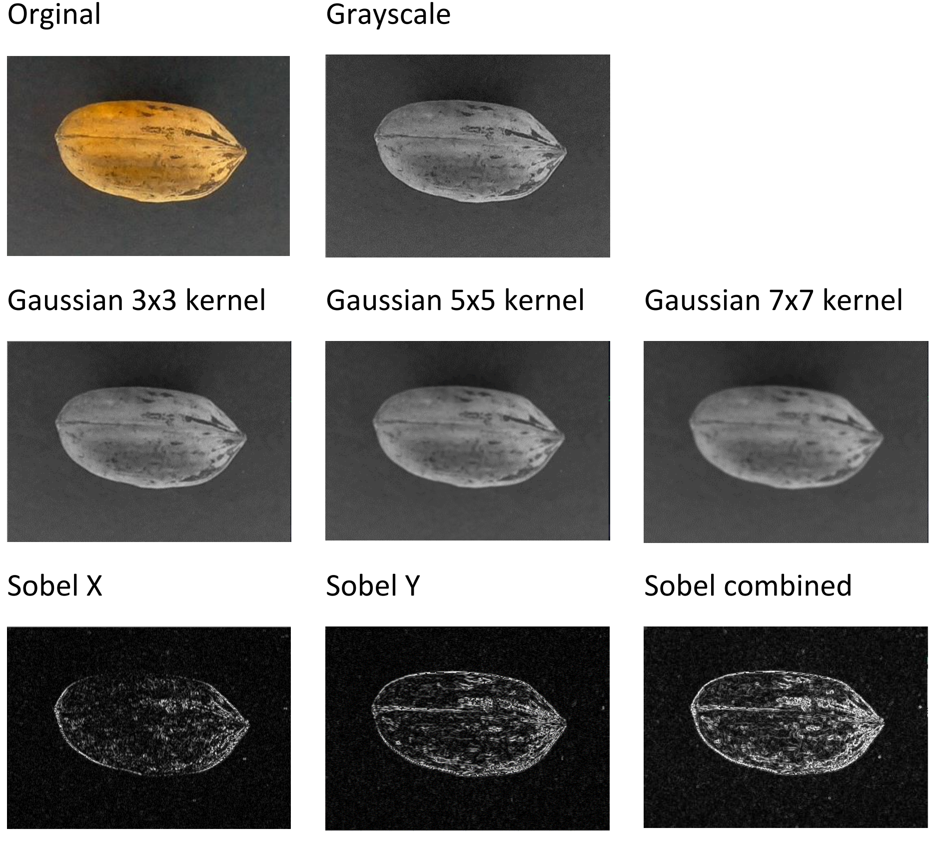


Figure 5-7 Blur and Edge detection with convolution

Figure 5-7 illustrates the effect a convolutional operation has on an image. As seen above, the convolution operation with different kernels creates different outputs. Starting from the top left with the original image of a Western Schley pecan nut. The image is converted to grayscale to aid the edge detection process. The middle row shows when a Gaussian kernel is applied to the grayscale input image. When the size of the kernel matrix is increased from 3x3 to 7x7 the image is blurred more aggressively, as seen above the image becomes more blurred with the increase in kernel size.

The bottom row of the image illustrates how to detect edges in the input image. The Sobel X kernel detects vertical edges while Sobel Y kernel highlights the horizontal edges, both these outputs can be combined to as seen in Sobel combined (bottom right).

In this section, a brief explanation was given how a convolution operation works and how specific kernels can transform an image. The next section will look at different layer types in a convolution neural network.

5.3. Layer types :

The next section will look at the most used layers in a convolutional neural network, an overview is given, and an explanation of how they work are presented.

The three different Layer types are:

- Convolutional Layer (CONV)
- Activation (ACT)
- Pooling (POOL)
- Fully-connected (FC)

5.3.1. Convolutional layer (CONV)

The CONV layer is considered as the basic building block of a CNN. This layer consists of a set of K learnable filters, i.e. kernels. Each kernel, as described in the previous section, has a width, height and depth. The depth of the CNN layer or network is also known as the number of channels. As in the case of an image, the input layer is equal to the number of colour channels in the image, for an RGB image, the channels are three (Reg, Green, Blue).

After applying K filters to the input image, Each kernel produces a 2D output called an activation map. These activation maps are stacked on top of each other K deep.

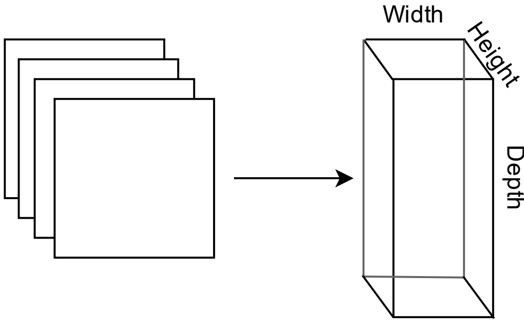


Figure 5-8 Activation Map (ROSEBROCK, 2017,p182)

There are three parameters which determine the output volume of a convolution layer, the depth (K), the stride (S) amount used and padding (P) used.

Depth (K) and Padding (P) has been discussed in the previous section. However, a new parameter stride (S) needs to be defined.

Stride is the number of steps the convolution process takes across and the input image. For a stride value of one, the kernel will move one pixel at a time, for a value of two the kernel will move two pixels.

By increasing the stride length, the spatial dimensions of an image are reduced. The output of an image is calculated by using eq 13:

$$\left(\frac{W - F + 2P}{S}\right) + 1 \tag{13}$$

Where :

W: Width of the square image

F: The receptive field, i.e. the kernel size

P: Amount of padding

S: Stride length

For example, For an input image of 224x224 pixels, Convoluted with a 3x3 Kernel, Zero Padding applied, and a stride of one, the output will be :

$$\left(\frac{224 - 3 + 2(1)}{1}\right) + 1 = 224$$

The CONV output will be 224 x 224 x K filters.

5.3.2. Activation (ACT)

A non-linear activation function, as described in section 4.1, is applied after every CONV layer. As the activation function performs an element-wise operation on the input volume, the output volume will be the size as the input dimension, $W_{input} = W_{output}$, $H_{input} = H_{output}$, $D_{input} = D_{output}$

5.3.3. Pooling (POOL)

As mentioned if the stride parameter is increased, the output will be reduced. However, there is another way to achieve this effect. A pooling layer is inserted after a CONV layer to reduce the spatial size of the input volume. By doing this, the number of parameters is also reduced. There is two POOL function used in neural networks which are max or average pooling.

Max pooling uses a pool size of 2x2. The block is slid across the input volume where the most significant value is kept before stepping with a stride length to the next pixels, as seen in Figure 5-9.

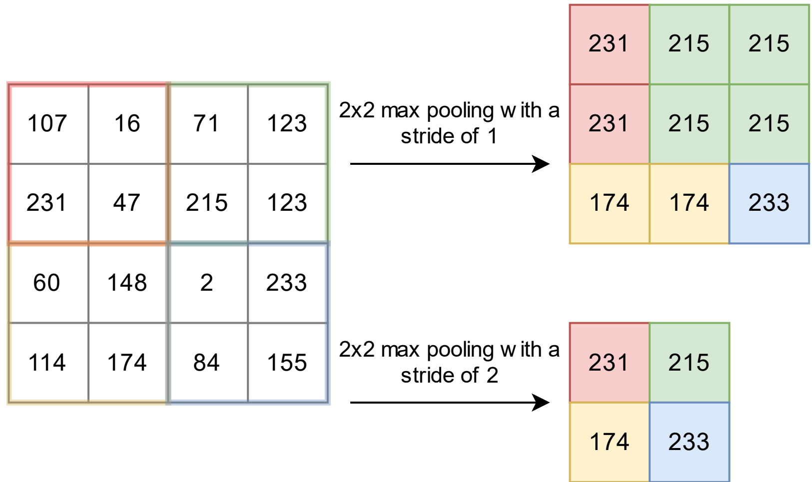


Figure 5-9 Max Pooling operation with different stride length (http://cs231n.stanford.edu/slides/2016/winter1516_lecture7.pdf)

With a stride length of two, the spatial dimension decreases drastically as seen above. Average pooling works the same as max pool, where instead of the maximum the average is taken of the block and kept as an output.

5.3.4. Fully-connected (FC)

The last layers in a neural network are the Fully Connected (FC) layers, as the name state they are fully connected to all the activations in the previous layer. The output of the FC layer is the input to the softmax classifier, which will compute the probabilities of each class in the dataset.

5.4. Loss functions :

The function that is used in the backpropagation algorithm to determine what the difference is between the actual output and the predicted output is called a loss function. The two commonly used loss functions in a backpropagation implementation are :

5.4.1. Mean Square Error (MSE) :

$$Loss(x, y) = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|^2 \tag{14}$$

The MSE function is a multiclass loss function that is used to determine the margin between the categories. Where x is a vector of n predictions, and y a zero initialised categorical binary vector where the element in the corresponding class is a 1.

5.4.2. Cross-Entropy :

Another multiclass loss function is called the Cross-Entropy loss, where the MSE loss gives the margin between the categories (classes) the Cross-Entropy gives you the probability of each class.

$$Loss(x, y) = -\sum_{i=1}^{n}\left[y_i \log\left(\frac{\exp(x_i)}{1 + \exp(x_i)}\right) + (1 - y_i) \times \log\left(\frac{1}{1 + \exp(x_i)}\right)\right] \tag{15}$$

The cross-entropy loss is favoured in convolutional neural networks. The reason is the loss function behaves more as one would expect to show the probability vs margin. Furthermore, the loss function speeds up the training because the error is more pronounced, i.e. the network converges more rapidly to a smaller loss where the MSE error tends to slow down the training because when the error is significant, it causes the derivative of the error to be small.

5.5. Optimisation algorithms

Optimisation algorithms are one of if not these most import element in machine learning. They are the engine that drives the learning process to learn the optimal weights and biases in a convolution neural network which will minimise the error in the prediction. The next section will look at how these algorithms work by taking the gradient descent algorithm as an example to illustrate how to minimise the error of a neural network.

5.5.1. Gradient descent

Gradient descent is an interactive algorithm that operates over an optimisation surface. The surface is depicted in Figure 5-10

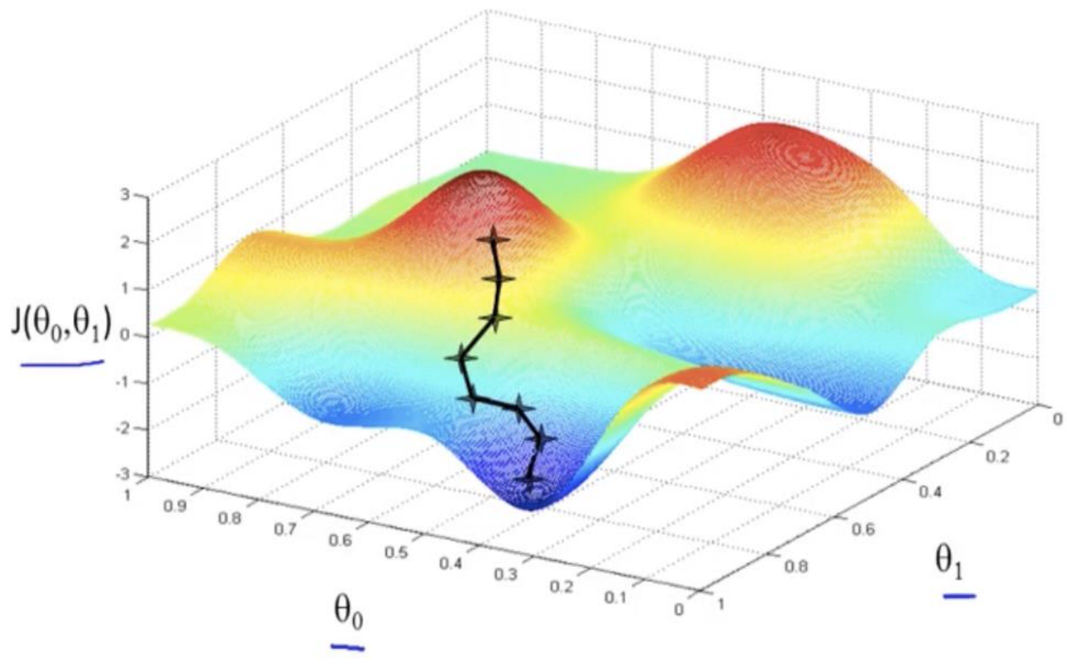


Figure 5-10 Gradient Descent (https://www.coursera.org/learn/machine-learning)

The red areas in the figure above indicate the global maximum errors from the cost function and the dark blue the global minimum errors. The gradient descent algorithm works like a ball rolling down a hill. The ball, in this case, the derivative of the error starts from a point on this error landscape. The idea is to find θ_0 and θ_1 which minimises the cost function to get to the global minimum. The next step in this process is to calculate the derivate of the error, by making use of eq 12 (section 4.2.2) to determine if the weights and biases need to be increased or decreased to minimise the error. Each of these steps theoretically brings the error to the global minimum, however on the way the function could get stuck in a local

minimum. The learning rate parameter defines the size each of these steps. If the learning rate is too small the algorithm could get stuck in a local minimum and other the other hand if the learning parameter is too large the algorithm will fail to converge to a global minimum point.

5.5.2. Stochastic Gradient Descent (SGD)

The Gradient descent, as described in the previous section, calculates the gradient on each point in the dataset, in a large dataset, this is prohibitively slow. One solution to this problem is to take small random samples from the dataset and bunch them together in a batch (Mini-Batch) and then update the weights and biases on the output of the batch called an Epoch. This method causes more noisy updates but has been proven to converge faster with no adverse side effects such as loss of Accuracy.

5.6. Regularisation approaches

To help prevent the neural network converge to only the training data in what is called overfitting, regularisation parameters are used. There are a few methods which could be used which will be briefly described below :

5.6.1. L2 Regularisation

With L2 regularisation, the cost function is penalised by adding a term. This prevents the network from modelling the training data precisely and help to generalise to new examples.

5.6.2. Data augmentation

Data augmentation is the process of rotating or scaling each example in the dataset to create more training examples artificially. This method causes that the network is unable to memorise all the examples and helps to show the network images which were not present in the original dataset, thus if the network encounters them in testing it can classify the input correctly.

5.6.3. Dropout

To prevent only specific nodes/neurons to become overactive in the learning process, a randomly selected set of nodes in the network are disconnect before each epoch. Srivastava et al. demonstrated how dropout could aid in addressing overfitting(Srivastava *et al.*, 2014).

5.6.4. Early stopping

A more straightforward method but still crucial to note is the act of merely stopping earlier in the training process.

5.7. Invariance

Convolutional Neural Networks can learn the needed invariance from the dataset. However, if the number of images is not significant enough, the model capability to generalise to unknown images are limited. In this section, three types of invariances are explained and suggestions made to limited the effect.

5.7.1. Rotation invariance

The CNN as a whole can be relatively tolerant to rotational invariance, but the individual filter layers need to learn how a specific object looks like when rotated. If the training dataset is scares of rotated images, and an unknown rotated image (test image) is presented, the network may fail to correctly classify the image or have a low probability in the specific class. With the aid of data augmentation, it is possible to generate the necessary rotated images for the filter layers to activate when an unknown image is presented.

5.7.2. Scale invariance

As with rotational invariance, the filter layers need to learn how an object looks when scaled. Data augmentation could be used to generate the necessary images to have enough data of the same object but scaled. Other methods are to train individual CNNs for each scale and combine their predictions.

5.7.3. Translation invariance

Translation invariance does not affect a CNN. The reason is that the convolution operation slide from left to right and top to bottom, where the filter will respond when coming across the edges, corners for example, irrespective where they are in the image. During the pooling operations, these responses dominate the neighbouring pixels by having more substantial activation. The network could be seen not to care where the activation is present but instead that it is present. With smaller dataset data augmentation could assist in making the network less sensitive for translation invariance by generating images which are moved to relative to the frame, i.e. not just centred in the frame.

5.8. Hierarchical feature learning

Before deep learning, the process of solving a machine learning problem was to handcraft features and only used the network to classify the images based on the features. With deep learning, the network has enough capacity to learn the features as part of the training process. However, this requires a significant amount of data. A.Rosebrock explains the difference in process with the following figure:

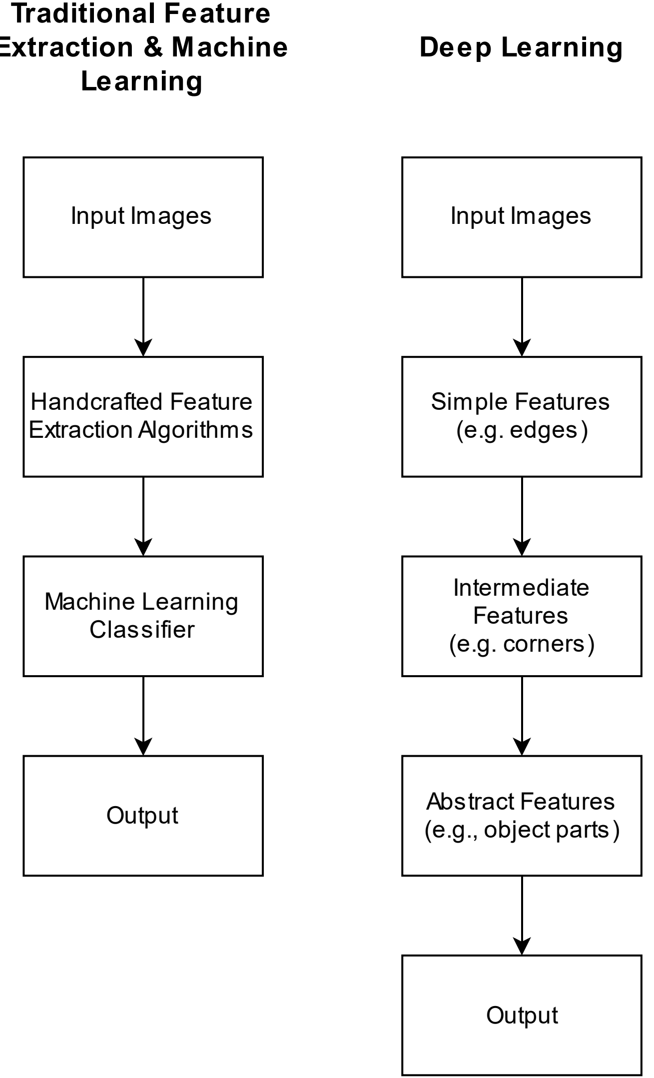


Figure 5-11 Traditional Feature Creation vs Deep Learning(ROSEBROCK, 2017)

In Figure 5-11 above the difference between the two methods, traditional feature extraction on the left and deep learning on the right is shown.

With the Traditional machine learning method, the human was responsible for defining specific features such as texture, shape, colour and train the network with these features. Where in Deep learning, the network learns the weights and biases of all the layers in the network, including the Convolution layers which were mentioned acts as features filters. A significant amount of data is required to propagate the errors back to the first layer in a deep neural network.

Because the network is able to learn the required features, the interpretation of these filters becomes difficult, as the network depth is increased the features becomes more abstract to represent and to make sense of visually.

5.9. Training methods

Convolution networks take a considerable time to train from an initialised state, also called from scratch. The network has to learn the weights and biases of all the layers could be in the order of a few million parameters. This process requires a significant dataset which consists of thousands of images, which is not always available. Recent research has shown that it is possible to transfer the features the network has learned to new unseen problems(Kaya *et al.*, 2019). The results are promising and opened up a new method for practitioners without the required data and hardware to build world-class accuracy classifiers. The two methods are briefly described next :

5.9.1. From Scratch:

When creating a new CNN, all the weights and biases in the network is randomly initialised. From these initialised state, the parameters need to be updated (learned) by making use of the backpropagation algorithm (section 4.2) and loss function (section 5.4) to predict and classify the dataset correctly. This process will yield the best results as this is a custom solution to the specific dataset. However, this approach has a few drawbacks. To train and update a few thousand and more likely millions of parameters, for example, in the case of the VGG-16 model, which has 140 Million parameters, takes signification amount of time with prohibitively expensive hardware. If the dataset is simple enough, which the typical image classification problem is not, then this approach could lead to excellent results with low-cost hardware.

Recent research has shown that machine learning problems can leverage the work that was done by the research community to solve new visual classification problems practically. This approach uses models such as VGG-16, which achieve a top 10% result in competitions like imagenet for new problems, without the need to train them from scratch. This method is called transfer learning which is discussed next.

5.9.2. Transfer learning:

Transfer learning is the process of using a pre-trained network such as MobileNetV2, VGG-16 or many others and removing the fully connected layers at the output of the network. A new fully connected (Dense) layers are added to the network which has the correct amount of classes required. The network is then trained on the training dataset where only the weights for the last layer is updated, and the rest of the network is kept the same. This procedure forces the pre-trained network to uses the pre-existing features learned to classify the new classes. Transfer learning is a two-step process; the first step is called :

5.9.2.1. Feature extracting

A pre-trained CNN consists of a convolutional base and a few fully connected layers to classify the input into the different classes. The feature extraction process is to replace the fully connected layers with new layers which will classify the new classes. The weights and biases in the convolutional base are kept intact(frozen) and used to extract features from the new dataset. These features are used to learn the new parameters of the fully connected layers. This process has the ability to train a model with typically 90% plus accuracy quickly. However, if higher accuracy is required, then the network needs to generalise better to the new dataset. To achieve this, a next step is performed call Fine-tuning.

5.9.2.2. Fine-tuning

The process of fine-tuning is when the top layers in the convolutional base are unfrozen and trained with the fully connected layers to further generalise better to the new dataset. With this approach, it is possible to achieve typically 95% plus accuracy, however one need to make use of regularisation approaches as discussed in section 5.6 to prevent overfitting to the new dataset.

5.10. Summary

In this section, the difference between a neural network and a CNN was explained. The convolutional layer, which consists of different filters to detect advanced features, were presented. The different effects the kernel matrix in convolutional operations has on an image were shown to illustrate how the filter is able to detect edges in an input image. Common architectures which are predominantly used in deep learning were explained. Then the working of a loss function and optimisation algorithms were presented to show their involvement in the learning process.

With the increase in depth of modern neural networks, their ability to generalise to any data has become a practical concern. To aid the network from overfitting to the training data regularisation methods need to be implemented, such as data augmentation and regularisations terms.

The ability of a CNN to be intolerant to different invariances were presented with methods to make the network more robust against variances in unseen data.

A CNN makes an excellent feature extractor which replaces the traditional handcrafted feature engineering process. Where in traditional machine learning implementations, the features need to be developed, which were a tedious process, and required specific domain knowledge to do correctly. Where in modern deep learning, the network is able to learn the required features directly from the data, with the caveat if there is enough data.

Different training methods were presented based on if a new model is developed and when a model will be reused and fine-tuned to the new dataset.

In the next chapter, an implementation process to complete a machine learning problem and the actual implementation of a CNN to classify different pecan nut cultivars will be discussed.

Chapter 6: Implementation of a Convolutional Neural Network.

The following chapter will describe the process followed to implement a CNN. They are:

- Hardware Implementation
- Capture and preparation of data
- Software Implementation

Figure 6-1 shows an overview of the Classification process. The pecan nut is captured by two CMOS cameras and transferred to the laptop for processing as depicted below. The detailed implementation is explained in the following sections.

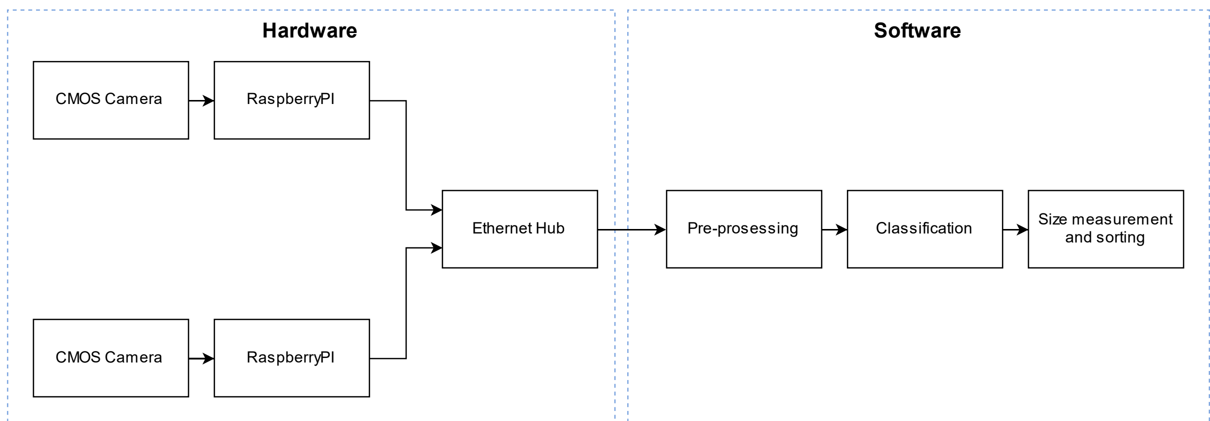


Figure 6-1 Classification Process flow

6.1. Hardware implementation

Figure 6-2 depicts the different elements of the hardware setup and Figure 6-3 show the actual hardware as used.

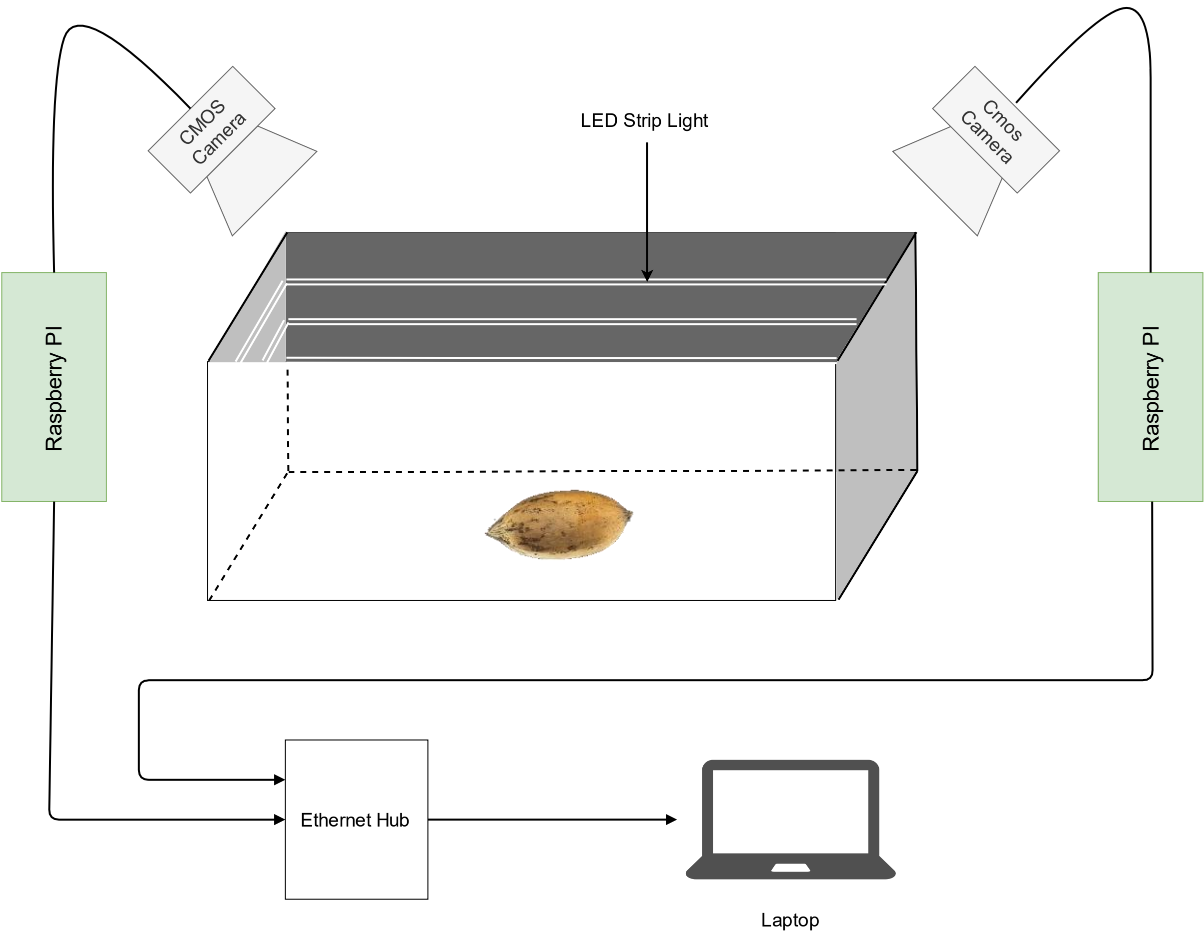


Figure 6-2 Hardware implementation overview

The different elements in the hardware setup are :

- A black box, which minimised shadows and attenuates the reflected light.
- LED light strip to provide even lighting.
- 2 x 5MP CMOS Cameras
- 2 x Raspberry Pi
- Ethernet Hub.
- Laptop computer.

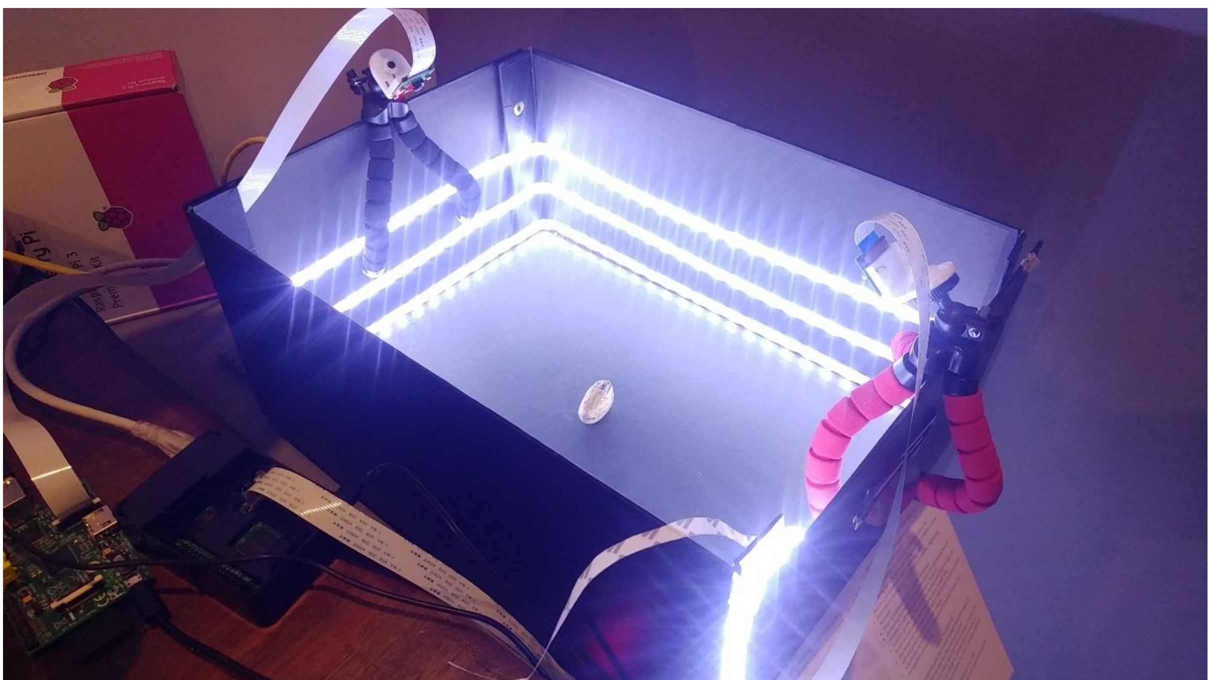


Figure 6-3 Actual Hardware

The two cameras capture the pecan nut image at a resolution of 800x600pixels. The images are then transferred in realtime to the laptop over an ethernet network, where the rest of the classification process happens. A custom application was developed for the raspberry pi to enable this image transfer.

The setup was inspired by what is typically used in commercial sorting machines. Figure 2-1 shows an example of such a setup, where the item falls through a gap surrounded by lights to minimise shadows and two cameras which takes an image from different angles on the same subject to increase the classification accuracy. In the setup, there are also Ejectors to reject the item into a reject receptacle. However, the mechanical removal of the pecan nut was considered outside of the scope of the research project, as the mechanical implementation is well known.

The next section will explain the data capturing process, which is vital to understand before the software processing are explained in the section after.

6.2. Data capturing and data pre-processing

There are no freely available datasets on pecan nuts which could be used for this research study, and thus a significant amount of effort and time was invested in capturing the required data for the project.

For the project, three different cultivars were chosen based on the amount of available pecan nut samples. The three cultivars are :

- Mahan
- Shoshoni
- Wichita.

These pecan nuts were produced in the northern cape province in South Africa, which is mostly a semi-arid region.

According to the Pecan Breeding & Genetics, Agricultural Service, U.S. Dept of Agriculture, the two different pecan nut cultivars are described as the following :

Table 6-1 Mahan, Shoshoni and Wichita pecan nuts

"Nut: oblong, with acute apex and base; nut often asymmetric, appearing 'pinched' in the middle due to flattening of abaxial and adaxial surfaces; flattened in cross-section; 32 nuts/lb, 58% kernel; kernels with deep secondary dorsal grooves and basal cleft, often poorly filled to base, woody in texture."	"Nut: oval elliptic with obtuse apex and rounded base; laterally compressed in cross-section; 41 nuts/lb, 53% kernel; kernels wrinkled with very wide dorsal grooves and deep basal cleft."
"Nut: oblong, with acute to acuminate, asymmetric apex and rounded apiculate base; round in cross-section; 43 nuts/lb, 62% kernel; kernels golden to light brown in colour with narrow dorsal grooves and a wide, shallow basal cleft."	

The captured dataset consisted out 495 individual pecan nuts of each cultivar, for each pecan nut are captured from two sides to make up a total of 990 images for each cultivar.

Each pecan nut was placed in the middle area of the black box as shown in Figure 6-2. The cameras captured the image and streamed the image to the laptop computer when the image appeared on the laptop screen a command was used to capture the images to the hard drive of the computer. To help with regularisation and to prevent the network from only seeing images from the same angle, which will cause poor testing accuracies. The pecan nut position was rotated clockwise every time a new nut was placed in the box, as seen below in Table 6-2.

Table 6-2 Rotation of pecan nut

Pecan Nut 1	Pecan Nut 2	Pecan Nut 3
Pecan Nut 4	Pecan Nut 5	Pecan Nut 6

6.2.1. Data pre-processing

Some experimentation has shown that it is necessary to remove the background from the images, this prevents the network from including the background in the training process, and thus affecting the accuracy of the network.

Table 6-3 shows the steps required to remove the background from the image. From the top left, the process is shown in 6 steps.

- Step 1: Shows the original image.

- Step 2: The image is converted to grayscale and blurred to create a clear boundary between subject and background
- Step 3: A canny edge detection algorithm is used to detect the edges.
- Step 4: The outside of the edges are detected by selecting the contour and fill.
- Step 5: From this image, a mask is built to remove the background from the original image.

The following section explains the program section, which does the removal. Snippets of the code are extracted and explained, and the key outputs will be shown, which will make it clear how the background removal process works. For the detailed source code see Appendix A.

Table 6-3 Remove background process

1.Original	2.Grayscale	3.Detect Edges
4.Find contours	5.Build Mask	6.Apply Mask

```
1. import imutils
2. from imutils import perspective

3. from imutils import contours
4. from imutils import resize

5. import cv2
6. import os

7. import numpy as np
```

LINE 1-7: imports the required libraries which were used. **LINE 1:** Imutils is a collection of image processing functions, and **LINE 5:** CV2 is the OpenCV library used to perform the necessary conversion and edge detection on the images.

```
8.  image = frameDict['picam-01']

9.  gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
10. gray = cv2.medianBlur(gray, 3, 0)
```

LINE 8-10: retrieves the image from the camera, converts the image to a grayscale image and blur the image with a median size 3x3 kernel. As seen in Table 6-3, step 1 and 2.

```
11. edged = cv2.Canny(gray, 40, 80)

12. edged = cv2.dilate(edged, None, iterations=1)
13. edged = cv2.erode(edged, None, iterations=1)
```

LINE 11-13: Detects all edges within a threshold range by using the canny edge detection algorithm. The dilate and erode function fills in neighbouring edges to form a continuous contour; this is useful to find complete the outside edge of pecan nut in the image. As seen in Table 6-3, step 3.

```
14. cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIM
    PLE)

15. cnts = imutils.grab_contours(cnts)
16.  (cnts, _) = contours.sort_contours(cnts)
17. index = np.argmax([np.shape(x)[0] for x in cnts])
```

LINE 14-17: Find all the edges that are linked to each other and define them as a contour. **LINE 17:** find the contour with the maximum size, which is the outside of the pecan nut. As seen in Table 6-3, step 4.

```
18. mask = cv2.drawContours(image.copy(), cnts, index, (0, 0, 0), -1)

19. mask[mask > 0] = 255
20. mask = cv2.bitwise_not(mask)

21. masked = cv2.bitwise_and(mask, image)
```

LINE 18-21: Makes a copy of the original image, draw the contour on the image by setting the inside of the contour area to zero (Black). **LINE19-20:** sets all the other areas which fall outside

the contour to 255 (White), and then inverts the mask to set the background to black and the pecan nut area to white. The next step is too logical-and the mask and the original image to mask out the background. As seen in Table 6-3, step 5 and 6.

This process is repeated for all the training, validation and test images before the model is trained. In the hardware use case, this pre-processing step is performed before the image is sent to the model for inference.

The last section described the hardware implementation, where the physical setup was shown and explained. The images are captured by using two cameras connected to raspberry pi's which streams the images to the laptop for pre-processing and classification. This same process was used to capture all the images for the training, validation and testing dataset, which is used in the next section, to train and evaluate the neural network.

This section will describe how the program work to train and evaluate a deep neural network on the laptop computer.

6.3. Software implementation

The next section will look at how the actual model was developed in TensorFlow. As described before convolution neural networks excel in image classification (Rawat and Wang, 2017); however, they require a significant amount of data (images) to achieve a decent accuracy. A different approach can be used to leverage the features a CNN has already learned and applied them to an unseen problem as described in section 5.9.2. The transfer learning process has achieved state-of-the-art results in image classification problems. Pan and Yang have provided the industry with a comprehensive review of transfer learning (Pan and Yang, 2010).

The following section describes how the transfer learning process was followed to successfully classify pecan nut cultivars on two different CNN's which has not been trained to identify pecan nuts. These two models are called :

- VGG16 from the Visual Geometry Group (VGG) at the University of Oxford.
- MobileNetV2 from Google.

Snippets of the code are extracted and explained, and the key outputs will be shown, which will make it clear how the implementation process was followed. For the detailed source code see Appendix A.

6.3.1. Training VGG16

```
1. import numpy as np
2. import tensorflow as tf
3. MAX_IMAGE_SIZE = 224
4. MAX_IMAGE_CHAN = 3
5. from tensorflow.keras.applications import VGG16
6. baseModel = VGG16(weights='imagenet',include_top=False,input_shape=(MAX_IMAGE_
   SIZE,MAX_IMAGE_SIZE,MAX_IMAGE_CHAN))
7. baseModel.summary()
```

LINE 1-2: imports the required libraries which were used in this research study.

LINE 3-4: set the maximum input image size and depth to 224x224x3 for and RGB image.

LINE 5: imports the VGG16 Model without the last classification layers attached, with transfer learning the last output layers need to be replaced with the categories relevant to the dataset, in this case, the three cultivars of pecan nuts. Figure 6-4 below shows the structure of the VGG16 neural network, which will form the base of the convolutional neural network.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Figure 6-4 VGG16 Convolutional base

The input image enters the network from the top at the layer named input_1, which is the Input layer without an activation function.

The next two layers are convolutional layers (CONV), each with the following variables :

$$O = Width\ of\ square\ output\ image$$

$$W = Width\ of\ square\ input\ image$$

$$\mathrm{S} = \text{Stride length}$$

$$F = \text{The receptive field, i.e. the kernel size}$$

$$\mathrm{P} = \text{Amount of padding}$$

According to eq 13, the output of the layer will be

$$O = \left(\frac{W - F + 2P}{S}\right) + 1$$

$$O = \left(\frac{224 - 3 + 2(1)}{1}\right) + 1 = 224$$

The number of parameters which need to be trained in each convolution layer is calculated with

$$W_c = Number\ of\ weights\ of\ the\ Conv\ Layer$$
$$B_c = Number\ of\ biases\ of\ the\ Conv\ Layer$$
$$P_c = Number\ of\ parameters\ of\ the\ Conv\ Layer$$
$$K = Size\ (width)\ of\ kernels\ used\ in\ the\ Conv\ Layer$$
$$N = Number\ of\ kernels$$
$$C = Number\ of\ \ channels\ of\ the\ input\ image$$

$$W_c = K^2\ \times C\ \times N$$
$$B_c = N$$
$$P_c = W_c + B_c$$

Number of Parameters for block1_conv1 Layer:

$$W_c = 3^2\ \times 3\ \times 64 = 1728$$
$$B_c = 64$$
$$P_c = 1728 + 64 = 1792$$

Number of Parameters for block1_conv2 Layer:

$$W_c = 3^2\ \times 64\ \times 64 = 63{,}864$$
$$B_c = 64$$
$$P_c = 63864 + 64 = 36{,}928$$

As the max-pooling operation does not introduce more neurons, there are no extra parameters to train. The output size of the layer is calculated with the same equation used for the convolutional operation but with zero paddings and stride length of 2:

$$O = \left(\frac{224 - 3 + 2(0)}{2}\right) + 1 = 111.5 = 112$$

In image processing, it is not possible to have half pixels. Hence the width needs to be rounded up to the nearest integer.

The next step is to set up the Image generators:

Image generators are iterators which reads the images from the hard drive in batches. These methods limit the amount of memory required for training or to put it differently, it enables the model to learn from large datasets as not all the images are kept in memory simultaneously.

LINE 8: Imports the OS library to build path string to directories.

LINE 9: Imports a custom library written by the author to automate repeating procedures.

```
8.  import os
9.  from ImageGenerators_Util import *
10. base_dir =...'\PiCamImages\Masked'
11. train_dir = os.path.join(base_dir,'train')
12. validation_dir = os.path.join(base_dir,'validation')
13. test_dir = os.path.join(base_dir,'test')

14. Train_gen,Val_gen,Test_gen = setup_ImageGenerators(train_dir,validation_dir,test_dir,
    ClassificationMode='categorical')
```

LINE 10-14: creates a path string to the correct images, used during the Training, Validation and testing phases.

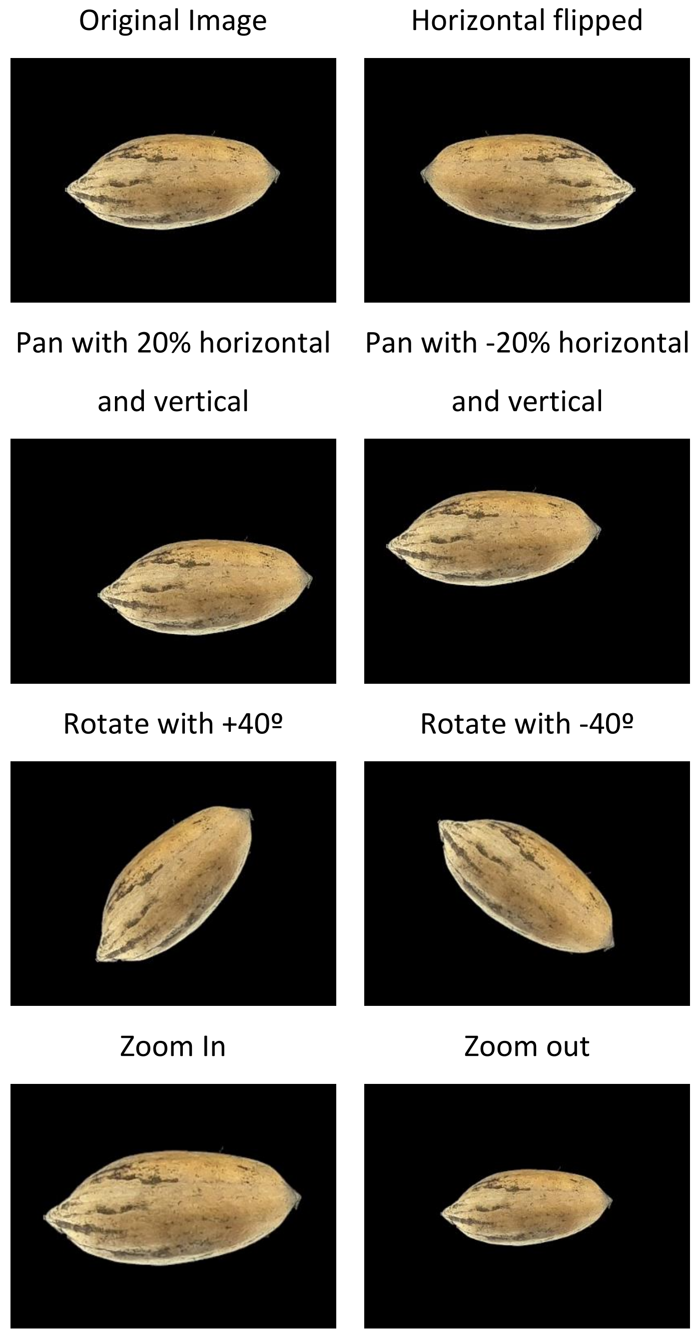
LINE 15-45: shows the setup_ImageGenerators function. This function set up each of the image generators for the training, validation and testing sets. The dataset is split in the following ratio :

Table 6-4 Dataset

	Mahan	Shoshoni	Wichita	Total	
Training	594	594	594	1782	60%
Validation	296	296	296	888	30%
Test	100	100	100	300	10%
				2970	100%

Besides reading the images from the hard drive, the function also does data augmentation (see section 5.6) where random images in a batch are either rotated with 40º, shifted horizontal and vertically with 20% or flipped horizontally as can be seen in Table 6-5. What this does it artificially increases the training dataset where one image is seen multiple times by the network but slightly adjusted in position, rotation or orientation. This effect as described in the previous section, also acts as regularisation to prevent overfitting to the training data.

Table 6-5 Data augmentation



Before the images are passed into the input layer of the network, it needs to be normalised as can be seen in **LINE 17,27,43.**

To further prevent overfitting to the data the images are shuffled in each batch, as the network repeatedly see the same images the order of the images can impact the update of the weights and cause the learning process to get stuck in a local minimum.

Each data generator classification mode is set to a categorical binary vector which will generate a label with each image as seen below :

$$Category\ 1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \ Category\ 2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \ Category\ 3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

and in the research problem case :

$$Mahan = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \ Shoshoni = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \ Wichita = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

```
15. def setup_ImageGenerators(Train_Dir,Validation_Dir,Test_Dir,
16.     ClassificationMode='categorical',MAX_IMAGE_SIZE = 224,MAX_IMAGE_CHAN=3):
17.     train_datagen = ImageDataGenerator( rescale=1./255,
18.                                 rotation_range=40,
19.                                 width_shift_range=0.1,
20.                                 height_shift_range=0.1,
21.                                 shear_range=0.2,
22.                                 zoom_range=0.2,
23.                                 horizontal_flip=True,
24.
25.                                 fill_mode='nearest')
26.
27.     validation_datagen = ImageDataGenerator(rescale=1./255)
28.
29.     train_generator = train_datagen.flow_from_directory(Train_Dir,
30.                 target_size=(MAX_IMAGE_SIZE, MAX_IMAGE_SIZE),
31.                                     batch_size=5,
32.                                     shuffle=True,
33.
34.                             class_mode=ClassificationMode)
35.
36.     validation_generator = validation_datagen.flow_from_directory(Validation_Dir,
```

```
37.                                 target_size=(MAX_IMAGE_SIZE, MAX_IMAGE_SIZE),
38.                                                 batch_size=5,
39.                                                 shuffle=True,
40.
41.                                         class_mode=ClassificationMode)
42.
43.     test_datagen = ImageDataGenerator(rescale=1./255)
44.     test_generator = test_datagen.flow_from_directory(Test_Dir,
45.                 target_size=(MAX_IMAGE_SIZE, MAX_IMAGE_SIZE),
46.                                         batch_size=1,
47.                                         shuffle=False,
48.                                 class_mode=ClassificationMode)
49.
50.     return train_generator,validation_generator,test_generator
```

The next step in the transfer learning method is to add a custom classification output layer/s to the output of the feature extractor network.

LINE 44-45: imports the layers and regularisation modules from the TensorFlow Keras library.

LINE 46-50: Adds two fully connected layers and dropout layer to the end of the convolutional base (baseModel), the flatten layer converts the (None,7,7,512) tensor to a (None,25088) vector. The None element will be replaced with the size of each batch during training and testing. The dropout layer is included to help with overfitting, and the output of the network is set to a softmax function. The softmax function will output the prediction of the network as a probability of each of the three classes.

```
51. from tensorflow.keras.layers import Dropout
52. from tensorflow.keras.layers import Flatten
53. from tensorflow.keras.layers import Dense
54.
55. # initialize the head model that will be placed on top of
56. # the base, then add a FC layer
57. headModel = baseModel.output
```

```
58. headModel = Flatten(name="flatten")(headModel)
59. headModel = Dense(D, activation="relu")(headModel)
60. headModel = Dropout(0.5)(headModel)
61.
62. # add a softmax layer
63. headModel = Dense(classes, activation="softmax")(headModel)
64.
65. model = Model(inputs=baseModel.input, outputs=headModel)
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 256)               6422784
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 771
=================================================================
Total params: 21,138,243
Trainable params: 21,138,243
Non-trainable params: 0
_________________________________________________________________
```

Figure 6-5 VGG16 Model architecture

Figure 6-5 shows the two networks stacked on top of each other. The Flatten layer is where the 7x7x512 tensor is transformed into a 1x25,088 vector.

The parameters for the fully connected layer (dense) between the final layer and the last convolutional layer is calculated by using the following formula:

$W_{cf} = Number\ of\ weights\ of\ a\ FC\ Layer\ which\ is\ connected\ to\ a\ Conv\ Layer$

$B_{cf} = Number\ of\ biases\ of\ a\ FC\ Layer\ which\ is\ connected\ to\ a\ Conv\ Layer$

$O = Size\ (width)\ of\ the\ output\ image\ of\ the\ previous\ Conv\ Layer$

$N = Number\ of\ kernels\ in\ the\ previous\ Conv\ layer$

$F = Number\ of\ neurons\ in\ the\ FC\ Layer$

$$W_{cf} = O^2 \times N \times F$$

$$B_{cf} = F$$

$$P_{cf} = W_{cf} + B_{cf}$$

Number of Parameters for dense Layer:

$$W_{cf} = 7^2 \times 512 \times 256 = 6{,}422{,}528$$

$$B_{cf} = 256$$

$$P_c = 6{,}422{,}528 + 256 = 6{,}422{,}784$$

The parameters for the fully connected layer (dense_1) is calculated by using the following formula:

$W_{ff} = Number\ of\ weights\ of\ a\ FC\ Layer\ which\ is\ an\ FC\ Layer$

$B_{ff} = Number\ of\ biases\ of\ a\ FC\ Layer\ which\ is\ connected\ to\ an\ FC\ Layer$

$P_{ff} = Number\ of\ parameters\ of\ a\ FC\ Layer\ which\ is\ connected\ to\ an\ FC\ Layer$

$F = Number\ of\ neurons\ in\ the\ FC\ Layer$

$F_{-1} = Number\ of\ neurons\ in\ the\ previous\ FC\ Layer$

$$W_{ff} = F_{-1} \times \mathrm{F}$$

$$B_{ff} = F$$

$$P_{ff} = W_{ff} + B_{ff}$$

Number of Parameters for dense_1 Layer:

$$W_{ff} = 256 \times 2 = 512$$

$$B_{ff} = 2$$

$$P_{ff} = 512 + 2 = 514$$

The total trainable parameters increased from 14.714 million to 21.138 million parameters, after the classification network was added. The next step in the transfer learning method is to fix convolutional base parameters that they are not updated during the training process, as seen in **LINE 66.**

```
66. baseModel.trainable = False
67. model.summary()
```

```
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 256)               6422784
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 771
=================================================================
Total params: 21,138,243
Trainable params: 6,423,555
Non-trainable params: 14,714,688
_________________________________________________________________
```

Figure 6-6 VGG16 Classification layer

This step decreases the trainable parameters from 21,138,243 to 6,423,555, which is significantly less to update, as seen in Figure 6-6. The second reason why this step is necessary is that the classification network has default initialised parameters, and the convolution base has already trained parameters. The magnitude of these parameters might differ significantly, and when the network is trained without setting the convolutional base trainable parameter to false, these new weights will destroy the VGG16 filter weights and biases in the update process.

The next step **LINE 68-74** is to set up the batch sizes and the number of epochs to train. As seen in **LINE 70,** the learning rate is fixed to $1e^{-3}$ for the initial training. The model is compiled in **LINE 76** with a categorical cross-entropy loss function and an RMS prop optimiser function.

The ‘Accuracy’ metric will be output from the training function as an indicator of how the model is progressing. **LINE 80-91** set up the callback functions required to save the training metrics into a file, which can be displayed later.

LINE 93 starts the training process, for the number epochs, the training batch sizes and the required validation step size.

```
68. num_train = len(Train_gen.filenames)
69. num_val = len(Val_gen.filenames)
70. learning_rate = 0.001
71. BATCH_SIZE = 32
72. num_epochs = 25
73. steps_per_epoch = round(num_train)//BATCH_SIZE
74.
75.
76. model.compile(loss='categorical_crossentropy',
77.         optimizer=tf.keras.optimizers.RMSprop(lr=learning_rate),
78.         metrics=['accuracy'])
79.


80. callbacks_list = [
81.         tf.keras.callbacks.TensorBoard(
82.                         log_dir=mylog_dir,
83.                         write_graph = False,
84.                         write_images = False,
85.                         update_freq = 'epoch',
86.                         profile_batch = 0,
87.                         embeddings_freq = 0,
88.                         embeddings_metadata = 0,
89.                         histogram_freq=1,
90.                         ),
91.         ]
```

```
92.
93. history = model.fit_generator(Train_gen,
94.                     steps_per_epoch=steps_per_epoch,
95.                     epochs=num_epochs,
96.                     callbacks=callbacks_list,
97.                     validation_data=Val_gen
98.                     )
```

After training for the required 25 epochs which took 18 minutes (average of 44 seconds per epoch), the validation accuracy went from 40% and started to converges to under 91%, as seen in Figure 6-7:

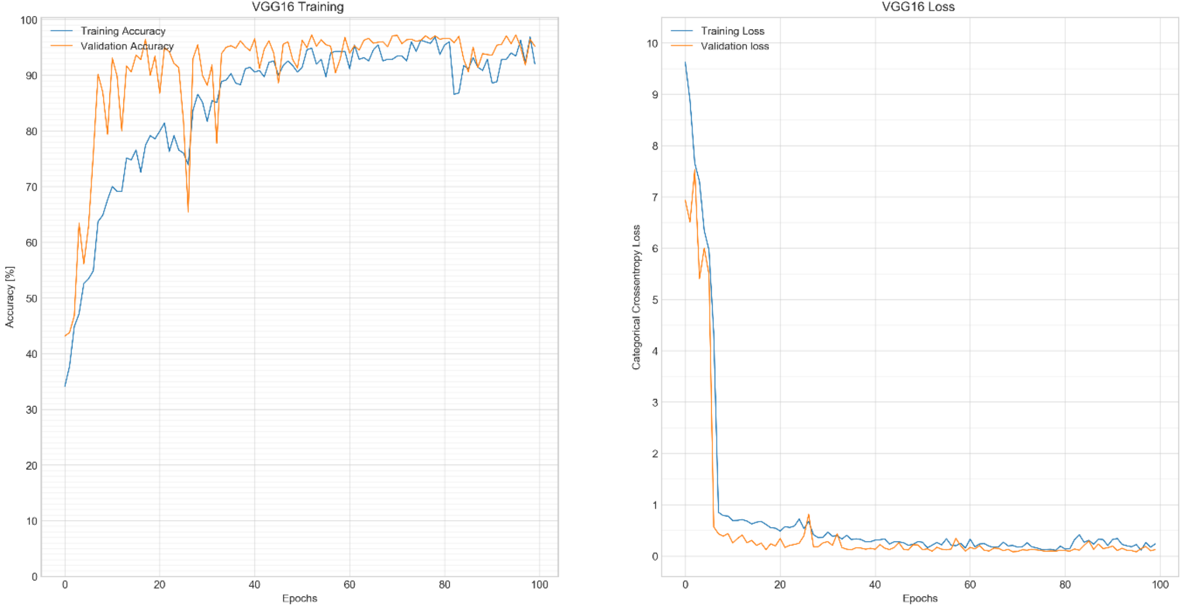


Figure 6-7 VGG16 Training and Validation

The next step is to fine-tune the model.

Fine-tuning is where some of the last convolutional layers in the base model are enabled for training, as shown in **LINE 99-100.**

```
99. for layer in baseModel.layers[15:]:
100.            layer.trainable = True
```

Figure 6-8 shows the number of parameters with the convolution layer BLOCK5 added to the trainable parameters.

```
block5_conv1 (Conv2D)        (None, 14, 14, 512)      2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)      2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)      2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)        0
_________________________________________________________________
flatten (Flatten)            (None, 25088)            0
_________________________________________________________________
dense (Dense)                (None, 256)              6422784
_________________________________________________________________
dropout (Dropout)            (None, 256)              0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                771
=================================================================
Total params: 21,138,243
Trainable params: 13,502,979
Non-trainable params: 7,635,264
_________________________________________________________________
```

Figure 6-8 VGG16 Fine-tuning training parameters

This step increases the trainable parameters from 6,423,555 to 13,502,979. The model is recompiled and the training function is called, but this time the optimiser are changed to SGD.

```
96. learning_rate = 0.001
97. model.compile(loss='categorical_crossentropy',
98.           optimizer=tf.keras.optimizers.SGD(lr=learning_rate),
99.           metrics=['accuracy'])
100.
101.        num_epochs = 25
102.        fine_tune_epochs = 50
103.        total_epochs =  num_epochs + fine_tune_epochs
104.
105.        history_fine = model.fit_generator(Train_gen,
106.                        steps_per_epoch=steps_per_epoch,
107.                        epochs=total_epochs,
108.                        initial_epoch=num_epochs,
109.                        callbacks=callbacks_list,
110.                        validation_data=Val_gen,
111.                        )
```

Within 50 additional epochs which took a further 36 minutes, the training and validation accuracy increased to 96%. A further convolutional block was added to the training parameters, and the learning rate decreased to $1e^{-4}$. The model was trained for another ten epochs where the accuracy stalled at 97% and the training was stopped to prevent overfitting. The accompanying loss graph (see Figure 6-7) also show the validation loss to be lower than the training loss, which means this model as successfully learned the required features to classify new unseen pecan nut images correctly.

6.3.2. Training MobileNetV2

The previous section described the transfer learning process, followed by using the VGG16 model. This section describes the transfer learning process, followed by using the MobileNetV2 model. The commands common to the two processes would be omitted to keep the section brief.

LINE 1-3 imports the required libraries to use in the model. **LINE 6** initialises the base model, which will with the weights set to the ImageNet values.

```
1. from tensorflow.keras.applications import MobileNetV2
2. from tensorflow.keras.layers import Dense
3. from tensorflow.keras.layers import GlobalAveragePooling2D
4.
5.
6. baseModel = MobileNetV2(weights="imagenet", include_top=False,
7.             input_tensor=Input(shape=(224, 224, 3)))
```

The output of the convolution base is shown in Figure 6-9, and the total number of parameters are shown in Figure 6-10.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1

Figure 6-9 MobileNetV2 Convolutional base

```
block_16_project (Conv2D)       (None, 7, 7, 320)    307200      block_16_depthwise_relu[0][0]

block_16_project_BN (BatchNorma (None, 7, 7, 320)    1280        block_16_project[0][0]

Conv_1 (Conv2D)                 (None, 7, 7, 1280)   409600      block_16_project_BN[0][0]

Conv_1_bn (BatchNormalization)  (None, 7, 7, 1280)   5120        Conv_1[0][0]

out_relu (ReLU)                 (None, 7, 7, 1280)   0           Conv_1_bn[0][0]
==================================================================================================
Total params: 2,257,984
Trainable params: 2,223,872
Non-trainable params: 34,112
__________________________________________________________________________________________________
```

Figure 6-10 MobileNetV2 Convolutional base parameters

As seen in Figure 6-10, the number of trainable parameters in the convolutional base alone is 2,223,872.

A classification network is attached to the base of the MobileNetV2 model. To convert the 7x7x1280 tensor into a 1x1280 vector, a global average pool and a dense layer were added. A softmax activation function is used for the last dense layer, which will give the probability of each of the three classes.

```
8.  headModel = baseModel.output
9.  headModel = GlobalAveragePooling2D()(headModel)
10. headModel = Dense(256, activation="relu")(headModel)
11.
12. headModel = Dense(len(Train_gen.class_indices), activation="softmax")(headModel)

13.
14. model = Model(inputs=baseModel.input, outputs=headModel)
```

```
block_16_project (Conv2D)         (None, 7, 7, 320)    307200      block_16_depthwise_relu[0][0]
__________________________________________________________________________________________________
block_16_project_BN (BatchNorma (None, 7, 7, 320)     1280        block_16_project[0][0]
__________________________________________________________________________________________________
Conv_1 (Conv2D)                   (None, 7, 7, 1280)   409600      block_16_project_BN[0][0]
__________________________________________________________________________________________________
Conv_1_bn (BatchNormalization)    (None, 7, 7, 1280)   5120        Conv_1[0][0]
__________________________________________________________________________________________________
out_relu (ReLU)                   (None, 7, 7, 1280)   0           Conv_1_bn[0][0]
__________________________________________________________________________________________________
global_average_pooling2d_1 (Glo (None, 1280)          0           out_relu[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                   (None, 256)          327936      global_average_pooling2d_1[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                   (None, 3)            771         dense_2[0][0]
==================================================================================================
Total params: 2,586,691
Trainable params: 2,552,579
Non-trainable params: 34,112
__________________________________________________________________________________________________
```

Figure 6-11 MobileNetV2 Classifier added

As seen in Figure 6-11, the total number of trainable parameters are 2,552,579 for the complete model.

The transfer process is followed where the convolution base training variable is set to false, to freeze the weights and biases during the training process. This reduces the number of trainable parameters to 328,707 (see Figure 6-12), which is significantly less than the 2.5 million parameters for the complete model.

```
Conv_1 (Conv2D)                   (None, 7, 7, 1280)   409600      block_16_project_BN[0][0]
__________________________________________________________________________________________________
Conv_1_bn (BatchNormalization)    (None, 7, 7, 1280)   5120        Conv_1[0][0]
__________________________________________________________________________________________________
out_relu (ReLU)                   (None, 7, 7, 1280)   0           Conv_1_bn[0][0]
__________________________________________________________________________________________________
global_average_pooling2d_1 (Glo (None, 1280)          0           out_relu[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                   (None, 256)          327936      global_average_pooling2d_1[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                   (None, 3)            771         dense_2[0][0]
==================================================================================================
Total params: 2,586,691
Trainable params: 328,707
Non-trainable params: 2,257,984
__________________________________________________________________________________________________
```

Figure 6-12 MobileNetV2 Convolutional base freeze

The model is compiled with an RMS prop optimiser function with the learning rate set to $1e^{-3}$ for the initial training. Because the output of the model is softmax activation function and the loss function is a categorical cross-entropy loss function which will give the probability of each class. The training function is executed to run for 25 epochs.

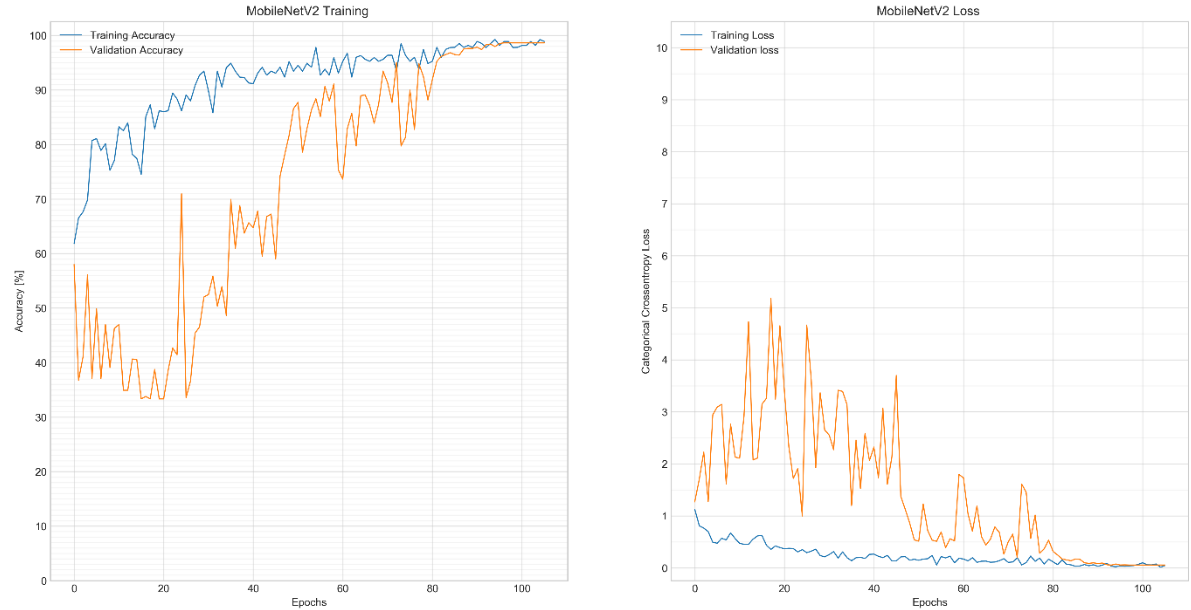


Figure 6-13 MobileNetV2 Training and Validation

As seen in Figure 6-13, the training accuracy increased from 63% to 88% in the first 25 epochs which took 14 minutes. However, the validation accuracy did not increase above 50%, which makes the accuracy of the model as good as a random guess. The observation could be made that the model is accurate on the training data but does not perform well on unseen data, such as the validation set. The model has what is called a high variance. High variance is when a model achieves a high training accuracy but low validation or testing accuracy. The learning rate was decreased to $1e^{-4}$ to see if the training process is overshooting the global minimum of the function. The model was trained for another 25 epochs were the accuracy increased to 65% and stalled again. The difference between the validation loss and training loss indicates that the model still has a high variance as one expects the loss validation loss to be better than the training loss. The model currently does not have enough depth to learn the new features required to classifies the pecan nut cultivars correctly. The next step in the transfer learning process was implemented, called fine-tuning. This is where some of the lower convolutional layers are unfrozen, and the model is able to update the weights of those layers to adjust the filters to adapt to the new images. The Block16 convolution block and the Conv_1, Conv_bn were added to the training parameters as seen below in Figure 6-14. The total number of trainable parameters increased from 328,707 to 1,214,787.

block_16_expand (Conv2D)	(None, 7, 7, 960)	153600	block_15_add[0][0]
block_16_expand_BN (BatchNormal	(None, 7, 7, 960)	3840	block_16_expand[0][0]
block_16_expand_relu (ReLU)	(None, 7, 7, 960)	0	block_16_expand_BN[0][0]
block_16_depthwise (DepthwiseCo	(None, 7, 7, 960)	8640	block_16_expand_relu[0][0]
block_16_depthwise_BN (BatchNor	(None, 7, 7, 960)	3840	block_16_depthwise[0][0]
block_16_depthwise_relu (ReLU)	(None, 7, 7, 960)	0	block_16_depthwise_BN[0][0]
block_16_project (Conv2D)	(None, 7, 7, 320)	307200	block_16_depthwise_relu[0][0]
block_16_project_BN (BatchNorma	(None, 7, 7, 320)	1280	block_16_project[0][0]
Conv_1 (Conv2D)	(None, 7, 7, 1280)	409600	block_16_project_BN[0][0]
Conv_1_bn (BatchNormalization)	(None, 7, 7, 1280)	5120	Conv_1[0][0]
out_relu (ReLU)	(None, 7, 7, 1280)	0	Conv_1_bn[0][0]
global_average_pooling2d_2 (Glo	(None, 1280)	0	out_relu[0][0]
dense_2 (Dense)	(None, 256)	327936	global_average_pooling2d_2[0][0]
dense_3 (Dense)	(None, 3)	771	dense_2[0][0]

```
Total params: 2,586,691
Trainable params: 1,214,787
Non-trainable params: 1,371,904
```

Figure 6-14 MobileNetV2 Fine-tuning training parameters

The model was recompiled but this time with a learning rate of 1e^{-5} and the optimiser set to an SGD function. As seen in Figure 6-13, the model was trained for another 50 epochs which took 28 minutes, where the accuracy increased from 65% to 98%. The accompanying loss graph also shows the validation loss to be lower than the training loss, which means this model successfully learned the required features to classify new unseen pecan nut images correctly.

6.3.3. Results

The previous section has explained in detailed how a VGG16 and MobileNetV2 model were implemented trained on a train and validation dataset and what accuracy has been achieved. The next step is to verify how well these models do with new unseen data. The snippets of source code used are extracted and explained, for the detailed source code see Appendix A.

6.3.3.1. Classification

As with the training of the model, the TensorFlow library has a test generator function which is used to read the images from the test dataset. The generator function also resize and normalise the images before running inference on them. **LINE 1-6** creates a list of predictions for each image in the test set and converts the label into a categorical vector. **LINE4** determine

if the prediction value is more than 50% for the category and if so, then set the value to a Boolean True value.

```
1. pred = model.predict_generator(Test_gen)
2. y_true = tf.keras.utils.to_categorical(Test_gen.classes, num_classes=3,
3.                                         dtype='bool')
4. y_pred = pred > 0.5
5. confusion_matrix = confusion_matrix(y_true.argmax(axis=1),
6.                                     y_pred.argmax(axis=1))
```

The test set consisted out of 100 images of each cultivar, representing 10% of the total dataset.

The confusion matrix plots the actual category against the predicted value, to show how accurate the model was to predict a specific cultivar.

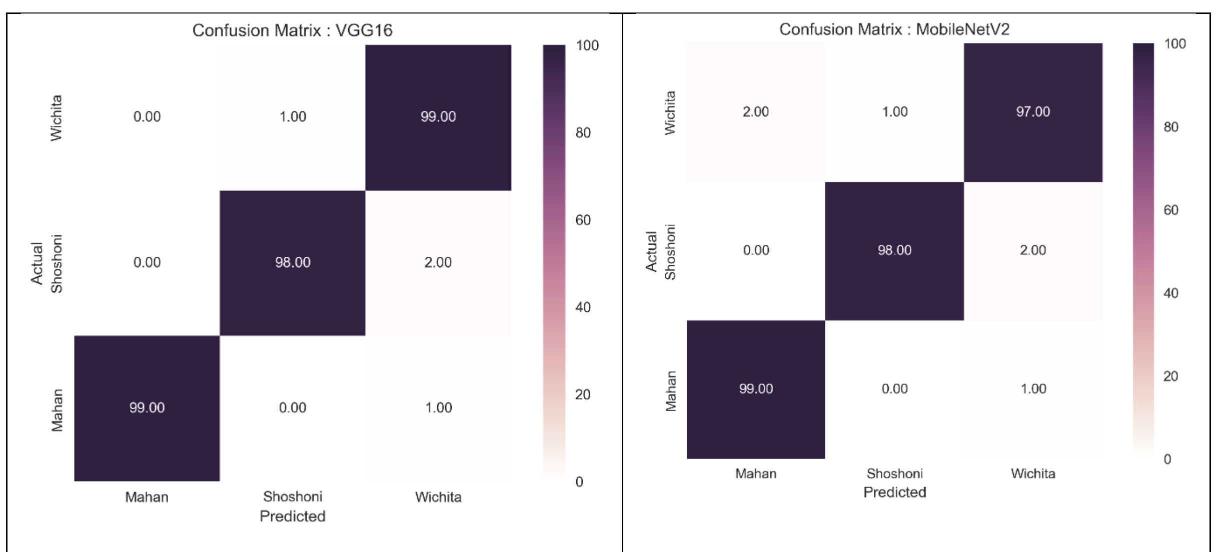


Figure 6-15 Confusion Matrix

As seen in Figure 6-15, both models did well to identify and classify the unseen images correctly. Both models had a 99% accuracy for the Mahan cultivar, a 98% accuracy for the Shoshoni. The Wichita had the most significant difference between the two models, where the VGG16 outperformed the MobileNetV2 with 2%. This difference is contributed to the fact that the VGG16 model is a wider model with more parameters (13.5M vs 1.2M parameters).

This is a factor of 11 times more trainable parameters and eight times more memory required to achieve a 2% increase in accuracy In one category. The memory use and parameters differences between the two models are shown in Table 6-6:

Table 6-6 Number of parameters and memory requirements

	VGG16	MobileNetV2
Number of parameters	13,502,979	1,214,787
Disk space and memory requirement	82,648 KB	10,565 KB

Both these models are suitable for an embedded environment; however, the MobileNetV2 with excellent accuracy is ideal for embedded devices like a raspberry PI, or mobile phones.

As in Figure 2-1 the practical implementation where the pecan nut freefall Infront of a camera system. The embedded device has less than a 100ms window available to make a prediction. The camera used has a framerate of 30fps which converts to a 33ms period per frame.

With modern convolutional neural such as the two models, the interpretability of how the model determines what type of pecan nut the input image is difficult, as the features maps are small (7x7 pixels in the last layer) it makes the presentation of how the model is activated during a prediction nearly impossible. Figure 6-16 shows such an image, as seen the lower layers are impossible to recognise visually. The darker areas in the images are where there were zero activations, i.e. dead filters.

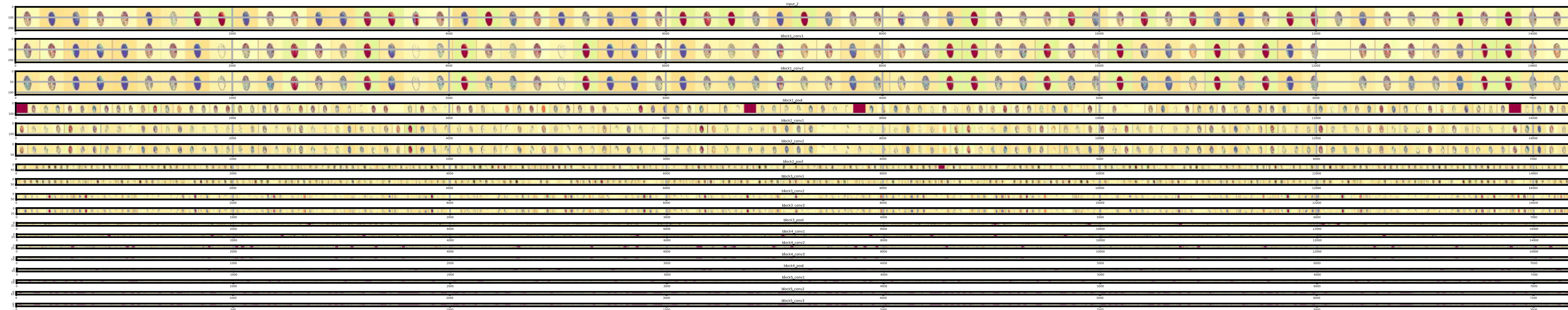


Figure 6-16 VGG16 Layer activation

However, it is possible to build a mask which shows which area contributed most to the prediction.

To understand better what area of an image is used the determine what the predicted output should be, an algorithm called Gradient-weighted Class Activation Mapping (Grad-CAM) is used to determine which features activated the model the most (Selvaraju *et al.*, 2020). Table 6-7 and Table 6-8 shows the difference between the three cultivars for each of the two models.

Table 6-7 VGG16 Grad-CAM

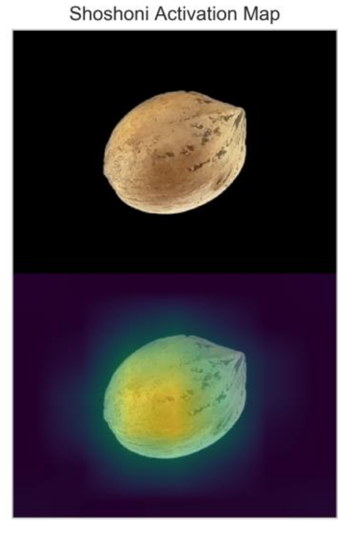
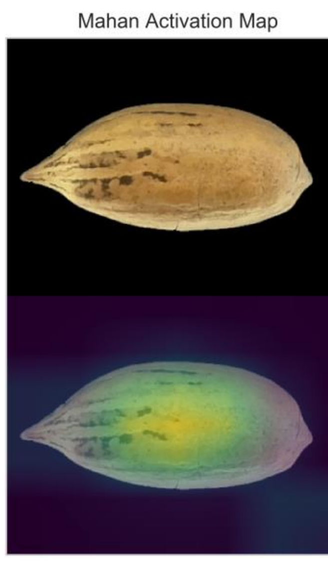
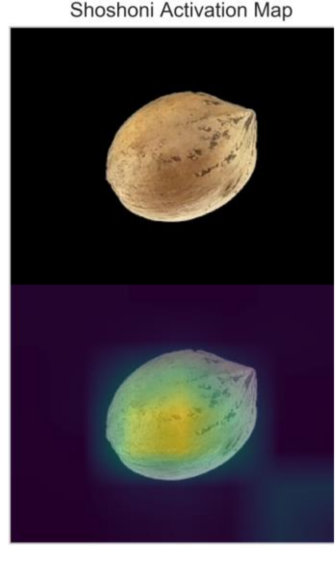
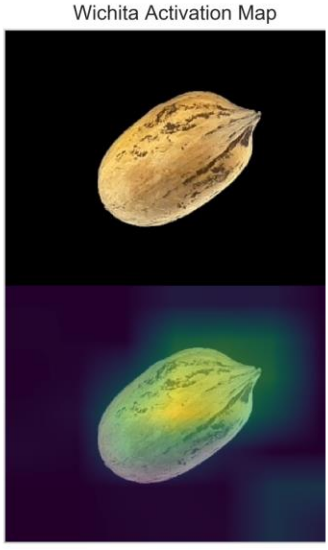


Table 6-8 MobileNetV2 Grad-CAM







6.3.3.2. Size measurements

With machine learning, one needs to define specific features to use in the model, with deep learning, the training process implements those features. One of such features could be the size. This section shows if the size of the pecan nuts in the images is determined what results would be obtained.

In Figure 6-17, starting from the left top, the major axis measurements in mm are given for the Mahan cultivar, in the top right, the semi-major axis measurements in mm are given. The middle row shows the measurements for the Shoshoni cultivar and the bottom row the measurements for the Wichita cultivar. With feature engineering one need to derive specific criteria which might be useful, for that reason, both camera's measurement is plotted and also the difference between the two measurements. In this case, the difference could be used to determine if a size measurement sample is valid. As the pecan nut is captured from to different angles, it is possible to have an incorrect measurement as there is no depth information capture with the images. As can be seen in Figure 6-17, the difference between the two cameras in the major axis is less than 5mm, where the difference in the semi-major axis is slightly higher.

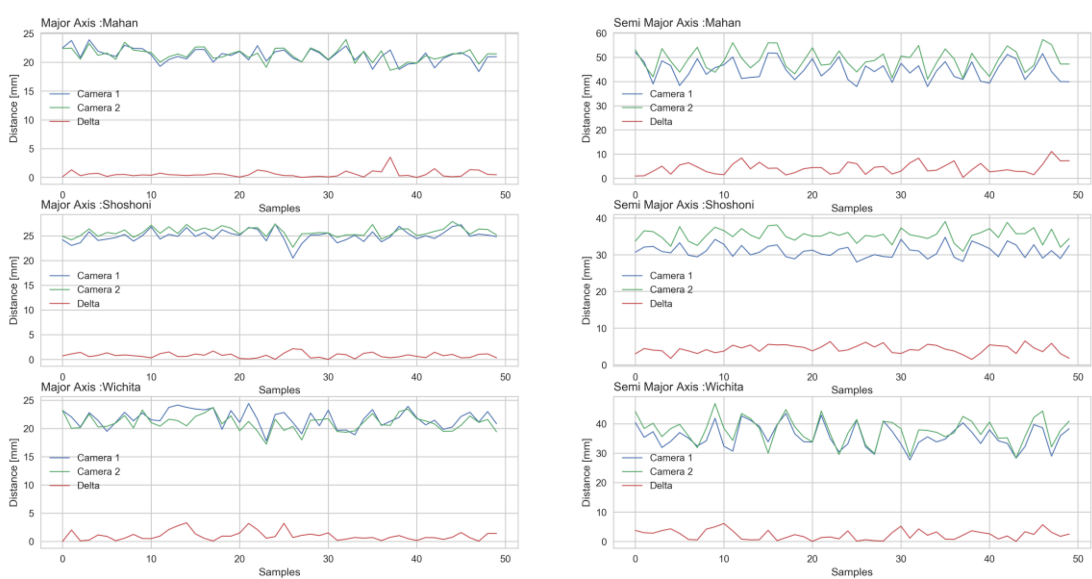


Figure 6-17 Size measurements

From the figure above it possible to see that if the two axes are used as criteria to determine the cultivar that one would be relatively successful in classifying the Mahan cultivar as the dimensions are suitable different from the other two cultivars, however, the classification accuracy would be lower on the other two cultivars as their dimension are almost similar.

6.3.3.3. Ratio measurements

Another feature which could be used would have been the ration between the major and semi-major axes, or the length and width or height of a pecan nut. As a pecan nut has three unique dimensions, length, width and height, it is not possible to determine in a particular image if the semi-major axis shows the height or width, so height is assumed. According to the Pecan Breeding & Genetics, Agricultural Service, U.S. Dept of Agriculture, the criteria to correctly determine a pecan nut cultivar according to dimension is a below.:

Descriptors for pecan nut shape based on nut length to height ratios are (repeated from section 3.1.2): .

- Orbicular 1 to 1.39
- Ovate 1.40 to 1.59, widest at base
- Obovate 1.40 to 1.59, widest at the apex
- Oval elliptic 1.40 to 1.59, widest in middle
- Elliptic 1.60 to 1.79
- Oblong elliptic1.80 to 1.99
- Oblong greater than 2.00

Descriptors of apex and base shape are very rudimentary;

- "acute" for angles sharper than 90 degrees
- "acuminate" for acute angles having concave surfaces; and
- "obtuse" for angles greater than 90 degrees.

Cross-section form is described as

- "round" if nut height to width ratios are between .95 and 1.10,
- "laterally compressed" if nut height to width ratios exceed 1.10, and as
- "flattened" if they are .95 or less.

And the definition for each of the cultivars in the research study is

- Mahan: oblong, with acute apex and base; nut often asymmetric, appearing 'pinched' in the middle due to flattening of abaxial and adaxial surfaces; flattened in cross-section
- Shoshoni: oval elliptic with obtuse apex and rounded base; laterally compressed in cross-section
- Wichita: oblong, with acute to an acuminate, asymmetric apex and rounded apiculate base; round in cross-section

Figure 6-18 shows the length to height ratios distribution of all the images in the dataset. According to the criteria supplied the Mahan cultivar ratio on both cameras matches with the specification. The Shoshoni cultivar is slightly lower than the specification on both cameras, and the Wichita cultivar considerably lower. The results could show if size measurements are essential, then more work should be put into characterising the camera setup and possible adding a method to measure the distance to the object to compensate for the field of depth.

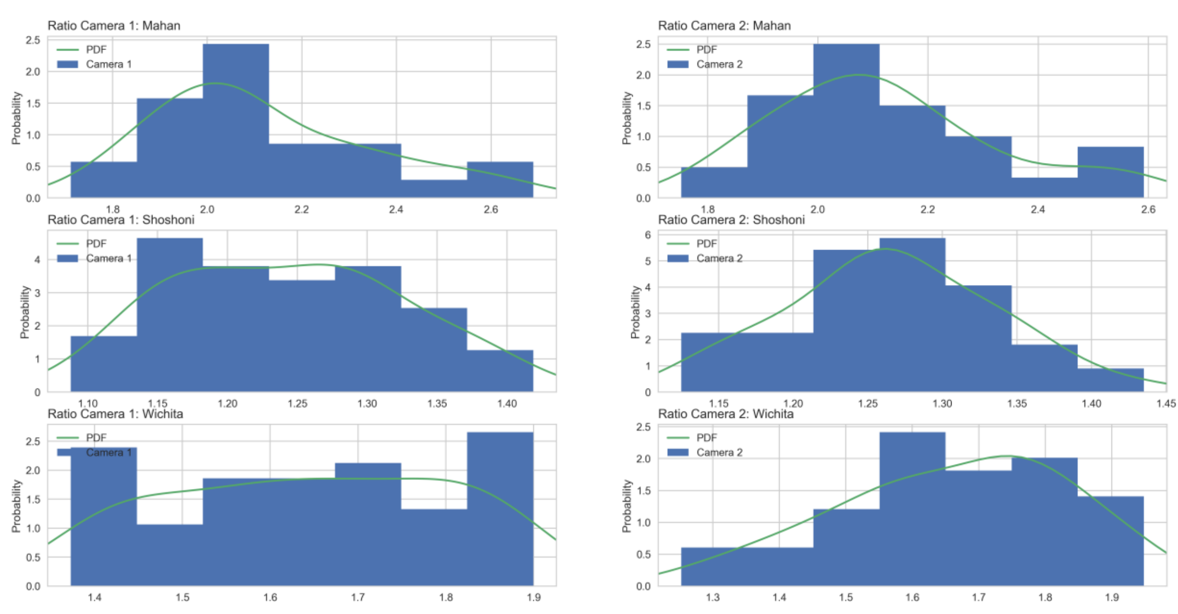


Figure 6-18 Length to Height ratios of samples

What is interesting to note is that the distribution mean value between the two camera's for each cultivar is quite similar. There is a slight overlap between the distributions of each

cultivar; however, one could use the ratio of a pecan nut in an image to make a decent prediction of the cultivar.

6.3.3.4. Colour measurements

The following section shows each cultivar colour spectrum and the combined spectrum of all the cultivars. The colour of a pecan nut could also be used as a feature to determine the cultivar of a pecan nut. Figure 6-19 to Figure 6-21 shows the colour RGB colour spectrum of each pecan nut cultivar. The Top row shows how the spectrum change over the images in the dataset and the bottom row shows the average and maximum values for each component of the RGB colour spectrum.

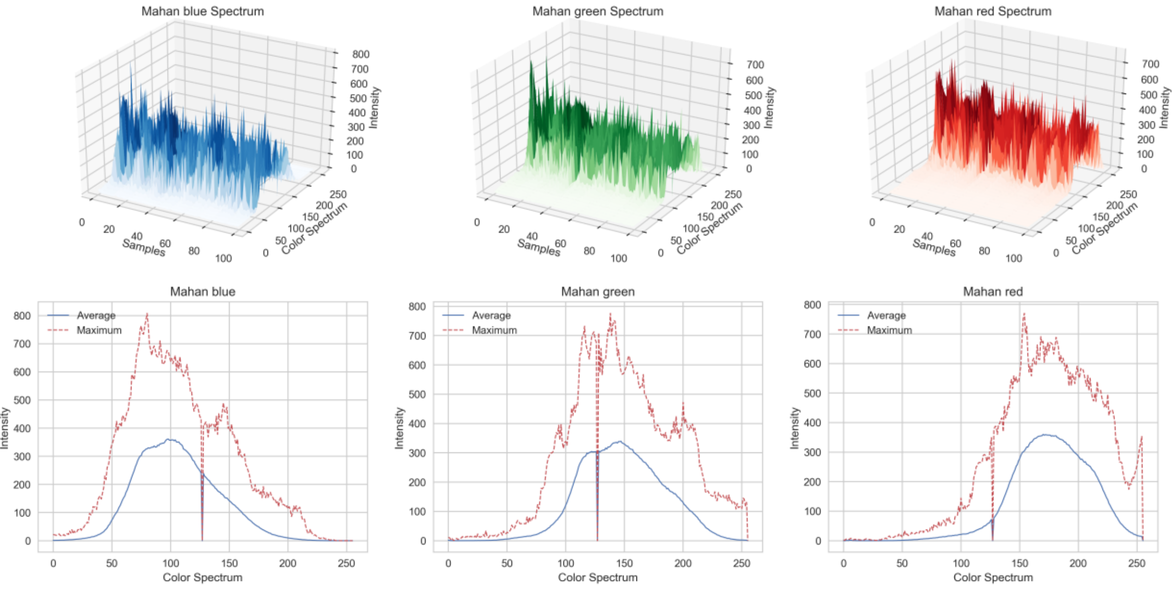


Figure 6-19 Mahan colour spectrum

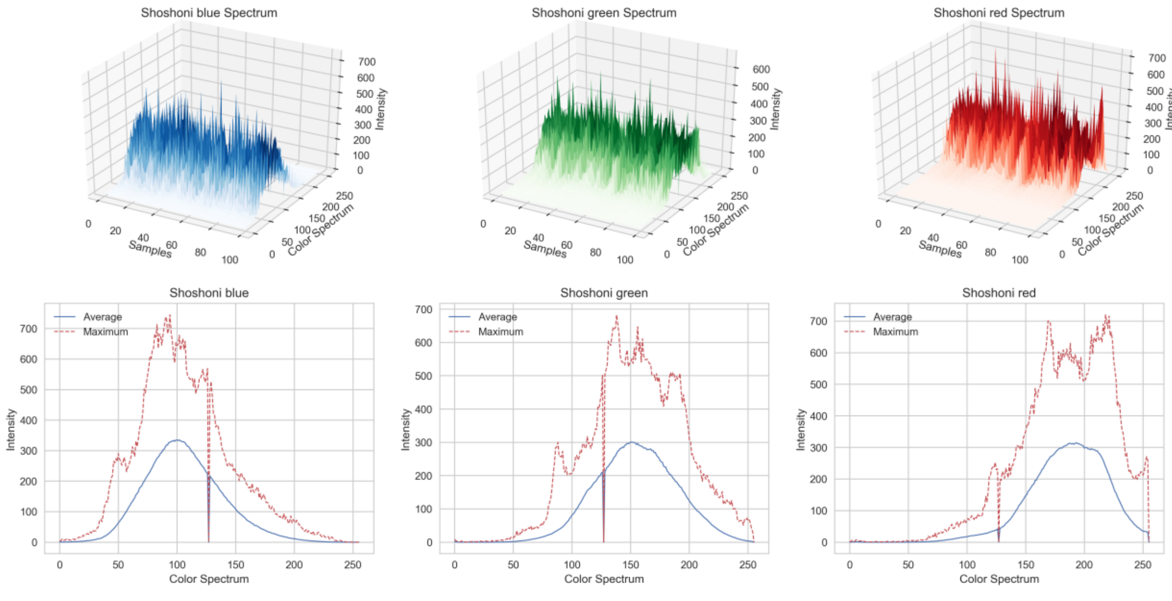


Figure 6-20 Shoshoni colour spectrum

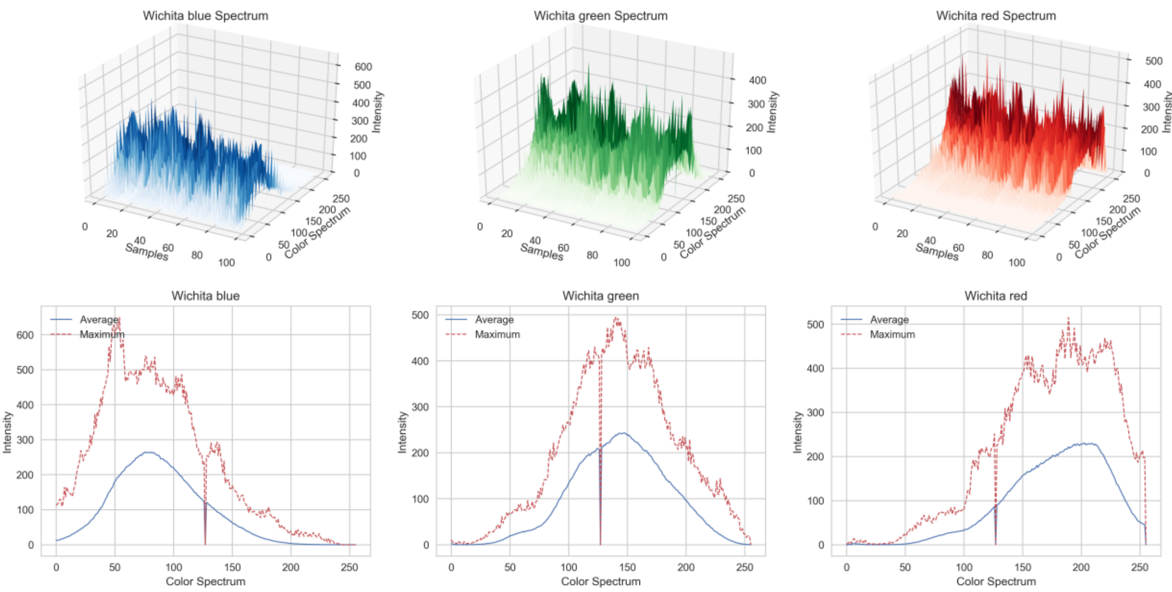


Figure 6-21 Wichita colour spectrum

Figure 6-22 shows all the different components for each cultivar overlaid to understand better how these values differ between the cultivars. It is interesting to note that there is quite a significant difference between the three cultivars regarding colour, which might make colour an excellent feature in a machine learning model.

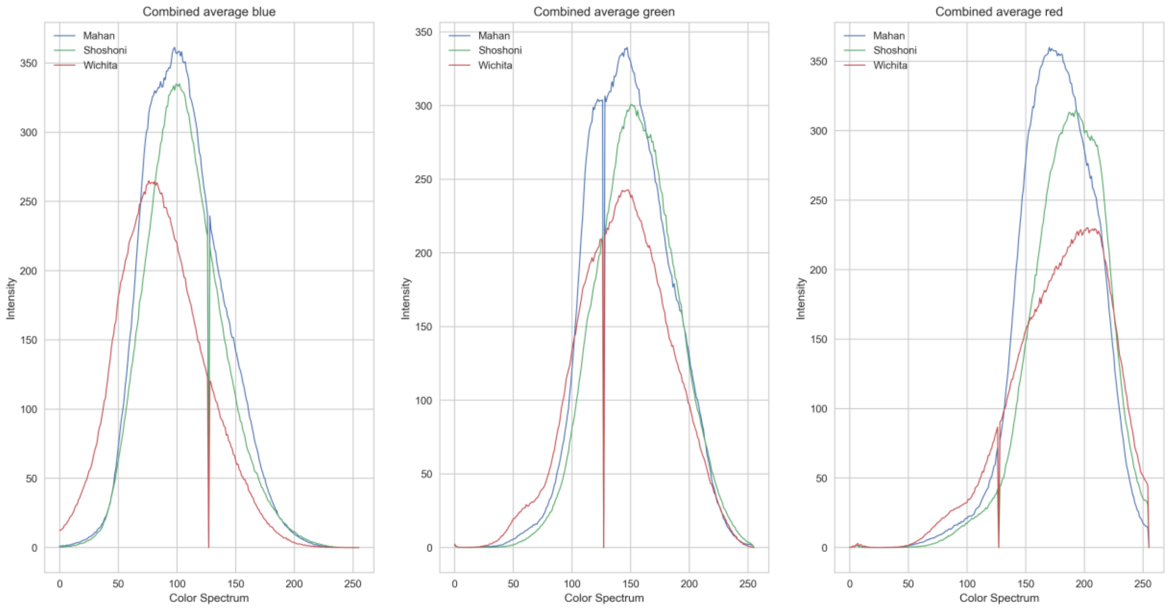


Figure 6-22 Combined Average and Maximum colour spectrum

6.4. Summary

The last section explained how two different convolution neural networks were implemented. The implementation was divided into a hardware and software section where the former showed how the physical hardware was built to capture the dataset of images and test the final model to confirm the accuracy. The software implementation section gave a detailed overview of the program developed and what each line of code does. The program made use of the TensorFlow library in python to create a neural network which consists of a convolutional base and classification network. The transfer learning process was followed to reuse the feature learned in the convolutional base to enable the classification network to identify and predict pecan nut cultivars previously not seen.

The complete training process was shown and how each network required a different strategy to achieve the required accuracy. The results were shown and discussed, and by making use of the Grad-Cam method to create an activation map examples of the areas which contributed the most to the prediction were shown.

The images in the dataset were analysed to verify what other features or parameters such as Size, ratio and colour are available which also could be used to make a prediction what type of cultivar is present in the image.

The next section will conclude the research study and present recommendations for future work.

Chapter 7:

7.1. Conclusions

Chapter 2 gave an overview of how a typical commercial sorting machine works and what it consists out of, and what research is currently being done in the field of agriculture and machine learning. According to the author, there are no other studies which use machine learning to classify pecan nut cultivars, which makes this research project a novel study. The chapter also showed the methodological approach to how the research will be completed and what each step entailed.

Chapter 3 gave the research methodology approach which were followed. The work done in the different work packages are presented with the software application which was used. A brief background were given regarding the software tools which were used such as TensorFlow, Numpy and OpenCV.

Chapter 4 gave a brief overview of the history of neural networks was discussed, and where machine learning and deep learning fit into the larger artificial intelligence field. The first neural network called a Perceptron were discussed and explained. Although important from a historical perspective, the algorithm had one major disadvantage, the inability to classify non-linear separable points.

For a machine-learning algorithm to handle more complex datasets, two elements are required :

- non-linear activation functions and a
- multi-layer network.

For a neural network to be able to learn the weights automatically, a backpropagation algorithm needs to be implemented, which consists of two phases :

1. The forward pass where an input propagates through the network to obtain the predicted output.
2. A backward pass where the gradient of the error is computed and the weights in individual nodes are updated by using the chain rule and the gradient descent algorithm.

Chapter 5 shown the difference between a neural network and a CNN. The convolutional layer, which consists of different filters to detect advanced features, were presented. The different effects the kernel matrix in convolutional operations has on an image were shown to illustrate how the filter is able to detect edges in an input image. Common architectures which are predominantly used in deep learning were explained. Then the working of a loss function and optimisation algorithms were presented to show their involvement in the learning process.

With the increase in depth of modern neural networks, their ability to generalise to any data has become a practical concern. To aid the network from overfitting to the training data regularisation methods need to be implemented, such as data augmentation and regularisations terms.

The ability of a CNN to be intolerant to different invariances were presented with methods to make the network more robust against variances in unseen data.

A CNN makes an excellent feature extractor which replaces the traditional handcrafted feature engineering process. Where in traditional machine learning implementations, the features need to be developed, which were a tedious process, and required specific domain knowledge to do correctly. Where in modern deep learning, the network is able to learn the required features directly from the data, with the caveat if there is enough data.

Different training methods were presented based on if a new model is developed and when a model will be reused and fine-tuned to the new dataset.

Chapter 6 explained how two different convolution neural networks were implemented. The implementation was divided into a hardware and software section where the former showed how the physical hardware was built to capture the dataset of images and test the final model to confirm the accuracy. The software implementation section gave a detailed overview of the program developed and what each line of code does. The program made use of the TensorFlow library in python to create a neural network which consists of a convolutional base and classification network. The transfer learning process was followed to reuse the feature learned in the convolutional base to enable the classification network to identify and predict pecan nut cultivars previously not seen.

The complete training process was shown and how each network required a different strategy to achieve the required accuracy. The results were shown and discussed and by making use of the Grad-Cam method to create an activation map examples of the areas which contributed the most to the prediction were shown.

The images in the dataset were analysed to verify what other features or parameters such as Size, ratio and colour are available which also could be used to make a prediction what type of cultivar is present in the image.

In response to the first investigative question posed as "What accuracy can be achieved by using a low accuracy camera and lens?", it was concluded that by making use either a VGG16 or MobileNetV2 model and transfer learning that an accuracy of 98% can be achieved. As the solution is aimed to automate the sorting process which is currently a manual process for many farmers, this accuracy should be compared against what a human typically can achieve, which is typically 85% (Toyofuku, Haff and Pearson, 2013, p. p237). The machine learning algorithm out preforms the manual process and is deemed a success.

In response to the second investigative question posed which asked "Can transfer learning be used to retrain a CNN successfully on pecan nuts?", it was concluded that transfer learning is not only successful but ideal for this solution. Transfer learning is a process where previously trained features are used as a base to build from to classify previous unseen categories, with a small dataset of new images.

In response to the third investigative question posed as" What type of pre-processing would improve accuracy?", it was concluded that by normalising the images and removing the background before training and running inference on an image the best results are achieved.

In response to the fourth and final investigative question posed as "What are other features available in the images?", it was concluded that both the ratio between length and height and the colour properties of each pecan nut are excellent features which could be used to determine the cultivar of a pecan nut.

This research has also significantly contributed to the machine learning research community by capturing a dataset of over 3000 images of pecan nuts for future research. More importantly, it produced a methodology to implement a working pecan nut classifier which

could be used on other projects for future research. The research is a first of a kind where pecan nut cultivars are classified with a low-cost optical system.

7.2. Recommendations

For future work, this method was developed as the start of a longer-term project to develop technology for the local agriculture industry. Industry 4.0 is disrupting the manufacturing industry, and this technology could be used to address the skill shortage currently facing in the agriculture industry. Future development can take this application to practical implementation on low-cost hardware, and test in a real-world environment.

An alternative algorithm could also be considered, such as SVM and Random forest, and evaluate them against the accuracy determined by using a CNN.

References

A. A. Gardea and M. A. Martínez-Téllez, R. C. for F. and, Development, Mexico and E. M. Yahia, A. U. of and Queretaro, M. (2011) *Postharvest biology and technology of tropical and subtropical fruits*. Woodhead Publishing Limited, 2011. Available at: https://www.sciencedirect.com/book/9780857090904/postharvest-biology-and-technology-of-tropical-and-subtropical-fruits.

BFAB (2018) *Agricultural Outlook 2018-2027*. Available at: https://www.oecd-ilibrary.org/docserver/agr_outlook-2016-10-en.pdf?expires=1528885846&id=id&accname=guest&checksum=58CBF2878A958890EAB90B72A2FCEA9B.

Bhargava, A. and Bansal, A. (2018) 'Fruits and vegetables quality evaluation using computer vision: A review', *Journal of King Saud University - Computer and Information Sciences*. doi: 10.1016/j.jksuci.2018.06.002.

Collet, F. (2018) 'Deep Learning with python', in *Deep Learning with python*.

F. Rosenblatt (1958) 'Rosenblatt solved the problem with his Perceptron', *Psychological Review*, 65(6), pp. 386–408.

Farmer's Weekly (2018) *Agricultural Outlook Spring Edition 2017/2018*.

Guggisberg, D. and Bosset, J. . (2003) 'Colour in food (Improving quality)'. Woodhead Publishing Limited, 2002, 36(3), pp. 116–142. doi: 10.1016/S0023-6438(02)00223-2.

Howard, A. G. *et al.* (2017) 'MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications'. Available at: http://arxiv.org/abs/1704.04861.

Kaya, A. *et al.* (2019) 'Analysis of transfer learning for deep neural network based plant classification models', *Computers and Electronics in Agriculture*. Elsevier, 158(October 2018), pp. 20–29. doi: 10.1016/j.compag.2019.01.041.

Kotwaliwale, N., Weckler, P. R. and Brusewitz, G. H. (2006) 'X-ray Attenuation Coefficients using Polychromatic X-ray Imaging of Pecan Components', *Biosystems Engineering*. doi: 10.1016/j.biosystemseng.2006.02.013.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2017) 'ImageNet Classification with Deep

Convolutional Neural Networks', *Commun. ACM*. New York, NY, USA: Association for Computing Machinery, 60(6), pp. 84–90. doi: 10.1145/3065386.

L. J. Grauke and T. E. Thompson (no date) *Pecan Breeding & Genetics, Agricultural Service, U.S. Dept of Agriculture*. Available at: https://cgru.usda.gov/carya/.

Lenny Wells and Patrick Conner (2015) 'Pecan Varieties for Georgia Orchards', *UGA Extension*, (Circular 898). Available at: https://secure.caes.uga.edu/extension/publications/files/pdf/C898_4.PDF.

Loukadakis, M., Cano, J. and O 'boyle, M. (2018) 'Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures', *11th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2018)*, (August). Available at: http://homepages.inf.ed.ac.uk/jcanore/pub/2018_multiprog.pdf.

Marvin Minsky and Seymour Papert (1970) 'A Review of "Perceptrons: An Introduction to Computational Geometry"', *INFORMATION AND CONTROL*, 17.

Mathanker, S. K. *et al.* (2011) 'AdaBoost classifiers for pecan defect classification', *Computers and Electronics in Agriculture*. doi: 10.1016/j.compag.2011.03.008.

Newell, A. and Simon, H. (2007) 'Computer science as empirical inquiry: symbols and search', in, p. 1975. doi: 10.1145/1283920.1283930.

Nielsen, M. A. (2015) *Neural Networks and Deep Learning*. Determination Press.

P. J. Werbos (1974) *Beyond regression : new tools for prediction and analysis in the behavioral sciences*. Harvard University.

Pan, S. J. and Yang, Q. (2010) 'A Survey on Transfer Learning', *IEEE Transactions on Knowledge and Data Engineering*, 22(10), pp. 1345–1359.

Pandey, R., Naik, S. and Marfatia, R. (2013) 'Image Processing and Machine Learning for Automated Fruit Grading System: A Technical Review', *International Journal of Computer Applications*. doi: 10.5120/14209-2455.

Rawat, W. and Wang, Z. (2017) 'Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review', *Neural Computation*, 29, pp. 1–98. doi: 10.1162/NECO_a_00990.

ROSEBROCK, A. (2017) *Deep learning for computer vision with Python: starter bundle.*

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986) ‘Learning representations by back-propagating errors’, *Nature*, 323(6088), pp. 533–536. doi: 10.1038/323533a0.

Sandler, M. *et al.* (2018) ‘MobileNetV2: Inverted Residuals and Linear Bottlenecks’, *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520. doi: 10.1109/CVPR.2018.00474.

Selvaraju, R. R. *et al.* (2020) ‘Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization’, *International Journal of Computer Vision*, 128(2), pp. 336–359. doi: 10.1007/s11263-019-01228-7.

Simonyan, K. and Zisserman, A. (2015) ‘Very deep convolutional networks for large-scale image recognition’, *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–14.

Srivastava, N. *et al.* (2014) ‘Dropout: A Simple Way to Prevent Neural Networks from Overfitting’, *Journal of Machine Learning Research*, 15, pp. 1929–1958. Available at: http://jmlr.org/papers/v15/srivastava14a.html.

Toyofuku, N., Haff, R. and Pearson, T. (2013) ‘10 - Advances in automated nut sorting A2 - Harris, Linda J’, in *Improving the Safety and Quality of Nuts*. doi: http://dx.doi.org/10.1533/9780857097484.2.230.

Warren S. McCulloch and Walter Pitts (1943) ‘A Logical calculus of the ideas immanent in nervous activity’, *Bulletin of Mathematical Biophysics*, 5, pp. 115–133.

Yann LeCun et al (1996) ‘Effiicient BackProp’, in *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*. London,UK: UK: Springer-Verlag.

Zeiler, M. D. and Fergus, R. (2013) ‘Visualizing and Understanding Convolutional Networks’.

Appendix A

```
1.  #================================= import required libaries =======================
    =================
2.  import tensorflow as tf
3.
4.
5.  #==================================================================================
    =================
6.
7.
8.  #================================= Set global parameters    =======================
    =================
9.  MAX_IMAGE_SIZE = 224
10. MAX_IMAGE_CHAN = 3
11.
12.
13. #==================================================================================
    =================
14.
15. #================================= Setup Image Generators    ======================
    ==================
16. import os
17. from ImageGenerators_Util import *
18.
19. #base_dir = 'D:\Johann\Workspace\Python\images_Pecan\images\Original_set'
20. base_dir = 'D:\Johann\Workspace\Python\images_Pecan\PiCamImages\Masked'
21. train_dir = os.path.join(base_dir,'train')
22. validation_dir = os.path.join(base_dir,'validation')
23. test_dir = os.path.join(base_dir,'test')
24.
25. Train_gen,Val_gen,Test_gen = setup_ImageGenerators(train_dir,validation_dir,test_di
    r,ClassificationMode='categorical')
26.
27. #==================================================================================
    =================
28.
29.
30.
31. #================================= Load Models    =================================
    =======
32. model = load_model('test_pecan_VGG16_3NUT_p98_Block4.h5') #==> VGG16 3 NUT 98%
33. model = load_model('test_pecan_MOBILENETV2_3NUT_p98b.h5') #==> MobilenetV2 3 NUT 98
    %
34.
35.
36.
37. #================================= Test and Plot confusion Matrix    ==============
    ==========================
38. from sklearn.metrics import confusion_matrix
39.
40. #test_loss, test_acc, test_mae = model.evaluate_generator(test_generator, steps=1)

41. test_loss, test_acc = model.evaluate_generator(Test_gen)
42. print('test acc:', test_acc)
43.
44. pred = model.predict_generator(Test_gen)
45. y_true = tf.keras.utils.to_categorical(Test_gen.classes, num_classes=3, dtype='bool
    ')
46. y_pred = pred > 0.5
47.
48. confusion_matrix_out = confusion_matrix(y_true.argmax(axis=1), y_pred.argmax(axis=1
    ))
49.
```

```
50. import seaborn as sn
51. import pandas as pd
52.
53. df_cm = pd.DataFrame(confusion_matrix_out,
54.   index = ["Mahan","Shoshoni","Wichita"],
55.   columns = ["Mahan","Shoshoni","Wichita"])
56.
57. fig = plt.figure()
58. plt.style.use('seaborn-whitegrid')
59.
60. plt.clf()
61. ax = fig.add_subplot(111)
62. ax.set_aspect(1)
63. cmap = sn.cubehelix_palette(light=1, as_cmap=True)
64. res = sn.heatmap(df_cm, annot=True, vmin=0.0, vmax=100.0, fmt='.2f', cmap=cmap)
65. res.invert_yaxis()
66. plt.yticks([0.5,1.5,2.5], ["Mahan","Shoshoni","Wichita"],va='center')
67. plt.title('Confusion Matrix : MobileNetV2')
68. plt.ylabel('Actual')
69. plt.xlabel('Predicted')
70. plt.savefig('confusion_matrix_mobileNetV2.png', dpi=300, bbox_inches='tight' )
71. plt.close()
72.
73. #=================================================================================
    =================
74.
75. #===================================  heatmaps  ===================================
    ============
76. from pyimagesearch.gradcam import GradCAM
77. import imutils
78.
79.
80. # initialize our gradient class activation map and build the heatmap
81. Pecan_nut_type = ["Mahan","Shoshoni","Wichita"]
82.
83. def output_heatmap(model_in,image_in,class_type_in):
84.     image = img_to_array(image_in)
85.     image = np.expand_dims(image, axis=0)
86.     image = image.astype("float") / 255.0
87.
88.     # the class label index with the largest corresponding probability
89.     preds = model_in.predict(image)
90.     i = np.argmax(preds[0])
91.     print(str(preds) + " - " + str(i))
92.     # initialize our gradient class activation map and build the heatmap
93.     cam = GradCAM(model, i)
94.     heatmap = cam.compute_heatmap(image)
95.
96.     # resize the resulting heatmap to the original input image dimensions
97.     # and then overlay heatmap on top of the image
98.     heatmap = cv2.resize(heatmap, (orig.shape[1], orig.shape[0]))
99.     (heatmap, output) = cam.overlay_heatmap(heatmap, orig, alpha=0.5)
100.
101.            return heatmap,output
102.
103.        for Pecan_index in range(0,3):
104.            base_dir = os.path.join('D:\Johann\Workspace\Python\images_Pecan\PiCamIm
    ages\Masked\Test')
105.            base_dir = os.path.join(base_dir,Pecan_nut_type[Pecan_index])
106.
107.            images_names = os.listdir(base_dir)
108.            test_img_files = [os.path.join(base_dir, f) for f in images_names]
109.
110.            for image_name,cnt in zip (test_img_files,range(10)):
111.                orig = cv2.imread(image_name)
112.                image = load_img(image_name, target_size=(224, 224))
```

```
113.                (heatmap, output) = output_heatmap(model,image,1)
114.                # display the original image and resulting heatmap and output image

115.                # to our screen
116.                output = np.vstack([orig, output])
117.                output = imutils.resize(output, height=700)
118.                output = cv2.cvtColor(output, cv2.COLOR_BGR2RGB)
119.
120.                fig = plt.figure()
121.                plt.style.use('seaborn-whitegrid')
122.                title = Pecan_nut_type[Pecan_index] +' Activation Map'
123.                plt.title(title)
124.                plt.grid(False)
125.                plt.imshow(output)
126.                plt.xticks([])
127.                plt.yticks([])
128.
129.                image_name = Pecan_nut_type[Pecan_index]+'_'+str(cnt)+'_Activation_M
    ap.png'
130.                print(image_name)
131.                plt.savefig(image_name, dpi=300)
132.                plt.close()
133.
134.        #============================================================================
    ========================
135.
136.
137.        #============================   determine distance/size =====================
    ========================
138.        from mpl_toolkits.mplot3d import Axes3D
139.        import re
140.        import imutils
141.
142.
143.        Pecan_index = 0
144.        Pecan_nut_type = ["Mahan","Shoshoni","Wichita"]
145.
146.        hist_blue_combine = np.zeros((256,0), dtype = "uint8")
147.        hist_green_combine = np.zeros((256,0), dtype = "uint8")
148.        hist_red_combine = np.zeros((256,0), dtype = "uint8")
149.
150.        dim_list = []
151.
152.        for Pecan_index in range(0,3):
153.            base_dir = os.path.join('D:\Johann\Workspace\Python\images_Pecan\PiCamIm
    ages\Masked\Test')
154.            base_dir = os.path.join(base_dir,Pecan_nut_type[Pecan_index])
155.
156.            images_names = os.listdir(base_dir)
157.            test_img_files = [os.path.join(base_dir, f) for f in images_names]
158.
159.            hist_blue = np.zeros((256,0), dtype = "uint8")
160.            hist_green = np.zeros((256,0), dtype = "uint8")
161.            hist_red = np.zeros((256,0), dtype = "uint8")
162.
163.            for img_path in test_img_files:
164.                image = cv2.imread(img_path)
165.                gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
166.                gray = cv2.medianBlur(gray, 7, 0)
167.                thresh = cv2.threshold(gray, 40, 255, cv2.THRESH_BINARY)[1]  #change
    d from gray
168.                thresh = cv2.erode(thresh, None, iterations=2)
169.                thresh = cv2.dilate(thresh, None, iterations=2)
170.                hist_blue = np.append(hist_blue,cv2.calcHist([image], [0], thresh, [
    256], [1, 255]),axis=1)
```

```
171.                hist_green = np.append(hist_green,cv2.calcHist([image], [1], thresh,
 [256], [1, 255]),axis=1)
172.                hist_red = np.append(hist_red,cv2.calcHist([image], [2], thresh, [25
6], [1, 255]),axis=1)
173.
174.                Camera = 0
175.                if(re.search("_01", img_path)):
176.                    Camera = 1
177.                    pixelsPerMetricA = 6.0
178.                    pixelsPerMetricB = 6.0
179.                else:
180.                    Camera = 2
181.                    pixelsPerMetricA = 6.5
182.                    pixelsPerMetricB = 6.0
183.
184.                cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,cv2.CHAIN_A
PPROX_SIMPLE) # changed from edge.copy.
185.                cnts = imutils.grab_contours(cnts)
186.                c = max(cnts,key=cv2.contourArea)
187.                im = np.zeros(shape=[MAX_IMAGE_SIZE, MAX_IMAGE_SIZE, 3], dtype=np.ui
nt8)
188.                ellipse = cv2.fitEllipse(c)
189.                (x,y),(MA,ma),angle = cv2.fitEllipse(c)
190.
191.                box = cv2.minAreaRect(c)
192.                box = cv2.cv.BoxPoints(box) if imutils.is_cv2() else cv2.boxPoints(b
ox)
193.                box = np.array(box, dtype="int")
194.                cv2.drawContours(image, [box], -1, (0, 255, 0), 2)
195.                # unpack the ordered bounding box, then compute the midpoint
196.                # between the top-left and top-right coordinates, followed by
197.                # the midpoint between bottom-left and bottom-right coordinates
198.                (tl, tr, br, bl) = box
199.                (tltrX, tltrY) = midpoint(tl, tr)
200.                (blbrX, blbrY) = midpoint(bl, br)
201.
202.                # compute the midpoint between the top-left and top-right points,
203.                # followed by the midpoint between the top-righ and bottom-right
204.                (tlblX, tlblY) = midpoint(tl, bl)
205.                (trbrX, trbrY) = midpoint(tr, br)
206.
207.                # compute the Euclidean distance between the midpoints
208.                dA_rect = dist.euclidean((tltrX, tltrY), (blbrX, blbrY))
209.                dB_rect = dist.euclidean((tlblX, tlblY), (trbrX, trbrY))
210.
211.                # if the pixels per metric has not been initialized, then
212.                # compute it as the ratio of pixels to supplied metric
213.                # (in this case, inches)
214.
215.                # compute the size of the object
216.                dimA_Rect = dA_rect / pixelsPerMetricA
217.                dimB_Rect = dB_rect / pixelsPerMetricB
218.
219.                if dimA_Rect > dimB_Rect:
220.                    tmp = dimB_Rect
221.                    dimB_Rect = dimA_Rect
222.                    dimA_Rect = tmp
223.
224.                print('rect : ' + str(dimA_Rect) + ' - ' + str(dimB_Rect))
225.
226.                dimA = MA / pixelsPerMetricA
227.                dimB = ma / pixelsPerMetricB
228.                print('eclipse : ' +str(dimA) + '-' + str(dimB))
229.
230.                dim_list.append([os.path.basename(img_path),Camera,MA,ma,angle,dimA,
dimB,dA_rect,dB_rect,dimA_Rect,dimB_Rect])
```

```
231.
232.
233.            hist  = [hist_blue,hist_green,hist_red]
234.            name_color = ["blue","green","red"]
235.            name_colormap = ["Blues","Greens","Reds"]
236.            rstride = 10
237.            cstride = 10
238.
239.            Y = range(256)
240.            X = range(100)
241.            X, Y = np.meshgrid(X, Y)
242.
243.            fig = plt.figure(figsize=(19.2,9.49))
244.            plt.style.use('seaborn-whitegrid')
245.
246.            for chan,cnt,color,colormap in zip(hist,range(1,4),name_color,name_color
    map):
247.                # set up the axes for the first plot
248.                ax = fig.add_subplot(2, 3, cnt, projection='3d')
249.                ax.plot_surface(X, Y, chan, rstride=rstride, cstride=cstride,
250.                                cmap=colormap, edgecolor='none')
251.                title = Pecan_nut_type[Pecan_index] +' ' + color + " Spectrum"
252.                ax.set_title(title)
253.                plt.legend(loc="upper left")
254.                ax.set_zlabel('Intensity')
255.                ax.set_ylabel('Color Spectrum')
256.                ax.set_xlabel('Samples')
257.
258.
259.
260.
261.            for chan,cnt,color,colormap in zip(hist,range(4,7),name_color,name_color
    map):
262.                # set up the axes for the first plot
263.                ax = fig.add_subplot(2, 3, cnt)
264.                ax.plot(np.average(chan,axis=1), color = 'C0',linestyle='solid',line
    width=1, label='Average')
265.                ax.plot(np.max(chan,axis=1), color = 'C2',linestyle='dashed',linewid
    th=1, label='Maximum')
266.                title = Pecan_nut_type[Pecan_index] +' ' + color
267.                ax.set_title(title)
268.                plt.legend(loc="upper left")
269.                ax.set_ylabel('Intensity')
270.                ax.set_xlabel('Color Spectrum')
271.
272.
273.            plt.savefig(Pecan_nut_type[Pecan_index]+'_colour_Spectrum'+'.png', dpi=3
    00)
274.            plt.close()
275.
276.            hist_blue_combine =  np.append(hist_blue_combine,hist_blue,axis=1)
277.            hist_green_combine =  np.append(hist_green_combine,hist_green,axis=1)
278.            hist_red_combine =  np.append(hist_red_combine,hist_red,axis=1)
279.
280.        fig = plt.figure(figsize=(19.2,9.49))
281.        plt.style.use('seaborn-whitegrid')
282.        hist  = [hist_blue_combine,hist_green_combine,hist_red_combine]
283.        for chan,cnt,color,colormap in zip(hist,range(1,4),name_color,name_colormap)
    :
284.            # set up the axes for the first plot
285.            print(cnt)
286.            print(color)
287.            print(colormap)
288.
289.            ax = fig.add_subplot(1, 3, cnt)
```

```
290.            ax.plot(np.average(chan[0:256,0:100],axis=1), color = 'C0',linestyle='so
    lid',linewidth=1, label=Pecan_nut_type[0])
291.            ax.plot(np.average(chan[0:256,100:200],axis=1), color = 'C1',linestyle='
    solid',linewidth=1, label=Pecan_nut_type[1])
292.            ax.plot(np.average(chan[0:256,200:300],axis=1), color = 'C2',linestyle='
    solid',linewidth=1, label=Pecan_nut_type[2])
293.            title = 'Combined average ' + color
294.            ax.set_title(title)
295.            plt.legend(loc="upper left")
296.            ax.set_ylabel('Intensity')
297.            ax.set_xlabel('Color Spectrum')
298.
299.        plt.savefig('Combined_colour_Spectrum'+'.png', dpi=300)
300.        plt.close()
301.
302.        tmp = np.zeros([150,12])
303.        idx = 0
304.
305.        for cnt in dim_list:
306.            print(cnt)
307.            if(cnt[1]==1): # if camera 1
308.                tmp[idx,0] = cnt[5] # Camera 1 DimA eclipse
309.                tmp[idx,1] = cnt[6] # Camera 1 DimB eclipse
310.                tmp[idx,2] = cnt[3]/cnt[2] # Camera 1 Ratio eclipse
311.
312.                tmp[idx,3] = cnt[9] # Camera 1 DimA rect
313.                tmp[idx,4] = cnt[10] # Camera 1 DimB rect
314.                tmp[idx,5] = cnt[10]/cnt[9] # Camera 1 Ratio rect
315.
316.            else: # if camera 2
317.                tmp[idx,6] = cnt[5] # Camera 2 DimA eclipse
318.                tmp[idx,7] = cnt[6] # Camera 2 DimB eclipse
319.                tmp[idx,8] = cnt[3]/cnt[2] # Camera 2 Ratio eclipse
320.
321.                tmp[idx,9] = cnt[9] # Camera 1 DimA rect
322.                tmp[idx,10] = cnt[10] # Camera 1 DimB rect
323.                tmp[idx,11] = cnt[10]/cnt[9] # Camera 1 Ratio rect
324.
325.                idx = idx + 1
326.
327.
328.        #================================    eclipse dimensions ====================
    ============
329.        fig = plt.figure(figsize=(19.2,9.49))
330.        plt.style.use('seaborn-whitegrid')
331.        section = 0
332.        cnt_plt = 1
333.        for cnt in range(3):
334.            print(cnt)
335.            print(cnt_plt)
336.            print(section)
337.            ax = fig.add_subplot(3, 2, cnt_plt)
338.            ax.plot(tmp[section:section+50,0], color = 'C0',linestyle='solid',linewi
    dth=1,label='Camera 1')
339.            ax.plot(tmp[section:section+50,6], color = 'C1',linestyle='solid',linewi
    dth=1,label='Camera 2')
340.            ax.plot(np.abs(tmp[section:section+50,0]-
    tmp[section:section+50,6]), color = 'C2',linestyle='solid',linewidth=1,label='Delta
    ')
341.            title = 'Major Axis :' + Pecan_nut_type[cnt]
342.            ax.set_title(title,loc='left')
343.            plt.legend(loc="center left")
344.            ax.set_ylabel('Distance [mm]')
345.            ax.set_xlabel('Samples')
346.
347.            ax = fig.add_subplot(3, 2, cnt_plt+1)
```

```
348.            ax.plot(tmp[section:section+50,1], color = 'C0',linestyle='solid',linewi
    dth=1,label='Camera 1')
349.            ax.plot(tmp[section:section+50,7], color = 'C1',linestyle='solid',linewi
    dth=1,label='Camera 2')
350.            ax.plot(np.abs(tmp[section:section+50,1]-
    tmp[section:section+50,7]), color = 'C2',linestyle='solid',linewidth=1,label='Delta
    ')
351.            title = 'Semi Major Axis :' + Pecan_nut_type[cnt]
352.            ax.set_title(title,loc='left')
353.            plt.legend(loc="center left")
354.            ax.set_ylabel('Distance [mm]')
355.            ax.set_xlabel('Samples')
356.
357.            section = section + 50
358.            cnt_plt = cnt_plt + 2
359.
360.        plt.savefig('Pecan_Dimension_eclipse.png', dpi=300)
361.        plt.close()
362.
363.        #========================================= eclipse dimensions ===============
    =================
364.
365.        #============================ rectangle  dimensions ========================
    ==================
366.        fig = plt.figure(figsize=(19.2,9.49))
367.        plt.style.use('seaborn-whitegrid')
368.        section = 0
369.        cnt_plt = 1
370.        for cnt in range(3):
371.            print(cnt)
372.            print(cnt_plt)
373.            print(section)
374.            ax = fig.add_subplot(3, 2, cnt_plt)
375.            ax.plot(tmp[section:section+50,3], color = 'C0',linestyle='solid',linewi
    dth=1,label='Camera 1')
376.            ax.plot(tmp[section:section+50,9], color = 'C1',linestyle='solid',linewi
    dth=1,label='Camera 2')
377.            ax.plot(np.abs(tmp[section:section+50,3]-
    tmp[section:section+50,9]), color = 'C2',linestyle='solid',linewidth=1,label='Delta
    ')
378.            title = 'Major Axis :' + Pecan_nut_type[cnt]
379.            ax.set_title(title,loc='left')
380.            plt.legend(loc="center left")
381.            ax.set_ylabel('Distance [mm]')
382.            ax.set_xlabel('Samples')
383.
384.            ax = fig.add_subplot(3, 2, cnt_plt+1)
385.            ax.plot(tmp[section:section+50,4], color = 'C0',linestyle='solid',linewi
    dth=1,label='Camera 1')
386.            ax.plot(tmp[section:section+50,10], color = 'C1',linestyle='solid',linew
    idth=1,label='Camera 2')
387.            ax.plot(np.abs(tmp[section:section+50,4]-
    tmp[section:section+50,10]), color = 'C2',linestyle='solid',linewidth=1,label='Delt
    a')
388.            title = 'Semi Major Axis :' + Pecan_nut_type[cnt]
389.            ax.set_title(title,loc='left')
390.            plt.legend(loc="center left")
391.            ax.set_ylabel('Distance [mm]')
392.            ax.set_xlabel('Samples')
393.
394.            section = section + 50
395.            cnt_plt = cnt_plt + 2
396.
397.        plt.savefig('Pecan_Dimension_rect.png', dpi=300)
398.        plt.close()
399.
```

```
400.        #============================ rectangle  dimensions ========================
    ==================
401.
402.
403.        #============================ eclipse  ratio ==============================
    ===========
404.
405.        import scipy.stats as st
406.
407.        fig = plt.figure(figsize=(19.2,9.49))
408.        plt.style.use('seaborn-whitegrid')
409.        section = 0
410.        cnt_plt = 1
411.        for cnt in range(3):
412.            print(cnt_plt)
413.            print(section)
414.        #Plot ratio of Mahan Camera 1
415.
416.            ax = fig.add_subplot(3, 2, cnt_plt)
417.            ax.hist(tmp[section:section+50,2], density=True, bins='auto', label="Cam
    era 1", linewidth=0.5,color='C0')
418.            mn, mx = plt.xlim()
419.            plt.xlim(mn, mx)
420.            kde_xs = np.linspace(mn, mx, 301)
421.            kde = st.gaussian_kde(tmp[section:section+50,2])
422.            ax.plot(kde_xs, kde.pdf(kde_xs), label="PDF",color='C1')
423.            plt.legend(loc="upper left")
424.            plt.ylabel('Probability')
425.            plt.title('Ratio Camera 1: '+ Pecan_nut_type[cnt],loc='left')
426.
427.            #Plot ratio of Mahan Camera 2
428.            ax = fig.add_subplot(3, 2, cnt_plt+1)
429.            ax.hist(tmp[section:section+50,8], density=True, bins='auto', label="Cam
    era 2", linewidth=0.5,color='C0')
430.            mn, mx = plt.xlim()
431.            plt.xlim(mn, mx)
432.            kde_xs = np.linspace(mn, mx, 50)
433.            kde = st.gaussian_kde(tmp[section:section+50,8])
434.            ax.plot(kde_xs, kde.pdf(kde_xs), label="PDF",color='C1')
435.            plt.legend(loc="upper left")
436.            plt.ylabel('Probability')
437.            plt.title('Ratio Camera 2: '+ Pecan_nut_type[cnt],loc='left')
438.
439.            section = section + 50
440.            cnt_plt = cnt_plt + 2
441.
442.        plt.savefig('Pecan_Ratios_eclipse.png', dpi=300)
443.        plt.close()
444.
445.        #============================ eclipse  ratio ==============================
    ===========
446.
447.        #============================ rectangle  ratio ============================
    =============
448.        fig = plt.figure(figsize=(19.2,9.49))
449.        plt.style.use('seaborn-whitegrid')
450.        section = 0
451.        cnt_plt = 1
452.        for cnt in range(3):
453.            print(cnt_plt)
454.            print(section)
455.            #Plot ratio of Mahan Camera 1
456.
457.            ax = fig.add_subplot(3, 2, cnt_plt)
458.            ax.hist(tmp[section:section+50,5], density=True, bins='auto', label="Cam
    era 1", linewidth=0.5,color='C0')
```

```
459.            mn, mx = plt.xlim()
460.            plt.xlim(mn, mx)
461.            kde_xs = np.linspace(mn, mx, 301)
462.            kde = st.gaussian_kde(tmp[section:section+50,5])
463.            ax.plot(kde_xs, kde.pdf(kde_xs), label="PDF",color='C1')
464.            plt.legend(loc="upper left")
465.            plt.ylabel('Probability')
466.            plt.title('Ratio Camera 1: '+ Pecan_nut_type[cnt],loc='left')
467.
468.            #Plot ratio of Mahan Camera 2
469.            ax = fig.add_subplot(3, 2, cnt_plt+1)
470.            ax.hist(tmp[section:section+50,11], density=True, bins='auto', label="Ca
    mera 2", linewidth=0.5,color='C0')
471.            mn, mx = plt.xlim()
472.            plt.xlim(mn, mx)
473.            kde_xs = np.linspace(mn, mx, 50)
474.            kde = st.gaussian_kde(tmp[section:section+50,11])
475.            ax.plot(kde_xs, kde.pdf(kde_xs), label="PDF",color='C1')
476.            plt.legend(loc="upper left")
477.            plt.ylabel('Probability')
478.            plt.title('Ratio Camera 2: '+ Pecan_nut_type[cnt],loc='left')
479.
480.            section = section + 50
481.            cnt_plt = cnt_plt + 2
482.
483.        plt.savefig('Pecan_Ratios_rect.png', dpi=300)
484.        plt.close()
485.        #=========================================================================
    ========================
486.
487.        #============================ image generators ============================
    =============
488.
489.        #from keras_preprocessing import image
490.        from keras_preprocessing.image import ImageDataGenerator,img_to_array,load_i
    mg
491.        from tensorflow.keras import backend as K
492.
493.        import matplotlib.pyplot as plt
494.
495.        import numpy as np
496.        import cv2
497.        from pathlib import Path
498.
499.        import os
500.
501.
502.
503.        def setup_ImageGenerators(Train_Dir,Validation_Dir,Test_Dir,ClassificationMo
    de='categorical',MAX_IMAGE_SIZE = 224,MAX_IMAGE_CHAN=3):
504.            train_datagen = ImageDataGenerator( rescale=1./255,
505.                                                rotation_range=40,
506.                                                width_shift_range=0.1,
507.                                                height_shift_range=0.1,
508.                                                shear_range=0.2,
509.                                                zoom_range=0.2,
510.                                                horizontal_flip=True,
511.                                                #validation_split=0.2,
512.                                                fill_mode='nearest')
513.
514.            validation_datagen = ImageDataGenerator(rescale=1./255)
515.
516.            train_generator = train_datagen.flow_from_directory(Train_Dir,
517.                                                                target_size=(MAX_IMA
    GE_SIZE, MAX_IMAGE_SIZE),
518.                                                                batch_size=5,
```

```
519.                                                               shuffle=True,
520.                                                               #class_mode='binary'

521.                                                               class_mode=Classific
    ationMode)
522.
523.           validation_generator = validation_datagen.flow_from_directory(Validation
    _Dir,
524.                                                                   target_size=(MAX
    _IMAGE_SIZE, MAX_IMAGE_SIZE),
525.                                                                   batch_size=5,
526.                                                                   shuffle=True,
527.                                                                   #class_mode='bin
    ary'
528.                                                                   class_mode=Class
    ificationMode)
529.
530.           test_datagen = ImageDataGenerator(rescale=1./255)
531.
532.           test_generator = test_datagen.flow_from_directory(Test_Dir,
533.                                                             target_size=(MAX_IMAGE
    _SIZE, MAX_IMAGE_SIZE),
534.                                                             batch_size=1,
535.                                                             shuffle=False,
536.                                                             class_mode=Classificat
    ionMode)
537.
538.           return train_generator,validation_generator,test_generator
539.       #===========================================================================
    ========================
540.
541.       #=========================== training and fine tuning of models ===========
    ===============================
542.       from tensorflow.keras.preprocessing.image import ImageDataGenerator
543.       from tensorflow.keras.optimizers import RMSprop
544.       from tensorflow.keras.optimizers import SGD
545.       from tensorflow.keras.applications import VGG16
546.       from tensorflow.keras.applications import MobileNetV2
547.       from tensorflow.keras.layers import Input
548.       from tensorflow.keras.models import Model
549.       from tensorflow.keras.callbacks import TensorBoard, EarlyStopping
550.       from imutils import paths
551.       import numpy as np
552.       import argparse
553.       import os
554.       from tensorflow.keras.models import load_model
555.
556.
557.       # import the necessary packages
558.       from tensorflow.keras.layers import Dropout
559.       from tensorflow.keras.layers import Flatten
560.       from tensorflow.keras.layers import Dense
561.
562.       class FCHeadNet:
563.           @staticmethod
564.           def build(baseModel, classes, D):
565.               # initialize the head model that will be placed on top of
566.               # the base, then add a FC layer
567.               headModel = baseModel.output
568.               headModel = Flatten(name="flatten")(headModel)
569.               headModel = Dense(D, activation="relu")(headModel)
570.               headModel = Dropout(0.5)(headModel)
571.
572.               # add a softmax layer
573.               headModel = Dense(classes, activation="softmax")(headModel)
574.
```

```
575.                # return the model
576.                return headModel
577. 
578. 
579.        #==================== VGG16 ======================
580.        baseModel = VGG16(weights="imagenet", include_top=False,
581.                          input_tensor=Input(shape=(224, 224, 3)))
582. 
583.        # initialize the new head of the network, a set of FC layers
584.        # followed by a softmax classifier
585.        headModel = FCHeadNet.build(baseModel, len(Train_gen.class_indices), 256)
586.        #headModel = FCHeadNet.build(baseModel, 1, 256)
587. 
588.        # place the head FC model on top of the base model -- this will
589.        # become the actual model we will train
590.        model = Model(inputs=baseModel.input, outputs=headModel)
591. 
592.        #==================== MOBILENET_V2 ======================
593. 
594.        from tensorflow.keras.layers import Dropout
595.        from tensorflow.keras.layers import Flatten
596.        from tensorflow.keras.layers import Dense
597.        from tensorflow.keras.layers import GlobalAveragePooling2D
598.        from tensorflow.keras import regularizers
599. 
600. 
601.        baseModel = MobileNetV2(weights="imagenet", include_top=False,
602.                          input_tensor=Input(shape=(224, 224, 3)))
603. 
604.        headModel = baseModel.output
605.        headModel = GlobalAveragePooling2D()(headModel)
606.        headModel = Dense(256, activation="relu")(headModel)
607. 
608.        headModel = Dense(len(Train_gen.class_indices), activation="softmax")(headMo
    del)
609. 
610.        model = Model(inputs=baseModel.input, outputs=headModel)
611. 
612. 
613.        #==============================================================
614.        for layer in model.layers:
615.            print(layer.name + " - " + str(layer.trainable))
616. 
617. 
618.        # loop over all layers in the base model and freeze them so they
619.        # will *not* be updated during the training process
620.        for layer in baseModel.layers:
621.            layer.trainable = False
622. 
623.        # compile our model (this needs to be done after our setting our
624.        # layers to being non-trainable
625.        print("[INFO] compiling model...")
626. 
627.        opt = RMSprop(lr=0.00001)
628.        #model.compile(loss="binary_crossentropy", optimizer=opt,
629.        #              metrics=["accuracy"])
630. 
631.        model.compile(loss="categorical_crossentropy", optimizer=opt,
632.                      metrics=["accuracy"])
633. 
634. 
635.        model.summary()
636. 
637. 
638.        #==================== VGG16 ======================
```

```
639.        model = load_model("D:\Johann\Workspace\Python\Tensorflow\Keras\\test_pecan_
    VGG16_2NUT.h5")
640.        #===================== VGG16 =======================
641.
642.        #===================== MOBILENET_V2 ======================
643.        model = load_model("D:\Johann\Workspace\Python\Tensorflow\Keras\\test_pecan_
    MobilenetV2_2NUT.h5")
644.        #===================== MOBILENET_V2 ======================
645.
646.        mylog_dir = 'mylogs\\2020031901\\run2'
647.
648.        callbacks_list = [
649.                    TensorBoard(
650.                                                        log_dir=mylog_dir,
651.                                                        write_graph = False,
652.                                                        write_images = False,
653.                                                        update_freq = 'epoch',
654.                                                        profile_batch = 0,
655.                                                        embeddings_freq = 0,
656.                                                        embeddings_metadata = 0,
657.                                                        histogram_freq=1,
658.                                                        )
659.                    ]
660.
661.
662.        print("[INFO] training head...")
663.        model.fit_generator(Train_gen,validation_data=Val_gen, epochs=25,steps_per_e
    poch=len(Train_gen.filenames) // 32,callbacks=callbacks_list)
664.
665.        # evaluate the network after initialization
666.        print("[INFO] evaluating after initialization...")
667.        predictions = model.predict_generator(Test_gen)
668.        predictions = model.predict(testX, batch_size=32)
669.        print(classification_report(testY.argmax(axis=1),
670.                                    predictions.argmax(axis=1), target_names=classNa
    mes))
671.
672.
673.        print(classification_report(testY.argmax(axis=1),
674.                                    predictions.argmax(axis=1), labels = labels))
675.
676.        #===================== VGG16 =======================
677.        # now that the head FC layers have been trained/initialized, lets
678.        # unfreeze the final set of CONV layers and make them trainable
679.        for layer in baseModel.layers[15:]:
680.            layer.trainable = True
681.
682.        for layer in baseModel.layers[11:]:
683.            layer.trainable = True
684.
685.        #===================== VGG16 =======================
686.
687.        #===================== MOBILENET_V2 ======================
688.        for layer in baseModel.layers[144:]:
689.            layer.trainable = True
690.
691.        for layer in baseModel.layers[135:]:
692.            layer.trainable = True
693.
694.        for layer in baseModel.layers[73:]:  #=> trying to solve the high val error,
     low training error issue.
695.            layer.trainable = True
696.
697.        #===================== MOBILENET_V2 =====================
698.
699.
```

```
700.        # for the changes to the model to take affect we need to recompile
701.        # the model, this time using SGD with a *very* small learning rate
702.        print("[INFO] re-compiling model...")
703.        opt = SGD(lr=0.00001)
704.        #model.compile(loss="binary_crossentropy", optimizer=opt,
705.        #              metrics=["accuracy"])
706.        model.compile(loss="categorical_crossentropy", optimizer=opt,
707.                      metrics=["accuracy"])
708.
709.
710.        # train the model again, this time fine-tuning *both* the final set
711.        # of CONV layers along with our set of FC layers
712.        print("[INFO] fine-tuning model...")
713.        model.fit_generator(Train_gen,validation_data=Val_gen, epochs=100,initial_ep
    och=81,steps_per_epoch=len(Train_gen.filenames) // 32,callbacks=callbacks_list, ver
    bose=1)
714.
715.
716.
717.        #=================================visuallise Training files ================
    ======
718.        from tensorboard.backend.event_processing import event_accumulator
719.        import matplotlib.pyplot as plt
720.
721.        Train_acc = np.zeros(0)
722.        Train_loss = np.zeros(0)
723.        Val_acc = np.zeros(0)
724.        Val_loss = np.zeros(0)
725.
726.        #run1
727.        trainFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs
    \\2020031901\\run1\\train\events.out.tfevents.1584642807.JOHANN-
    LAPTOP.25888.252226.v2")
728.        valFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs\\
    2020031901\\run1\\validation\events.out.tfevents.1584642807.JOHANN-
    LAPTOP.25888.252248.v2")
729.        #run2
730.        trainFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs
    \\2020031901\\run2\\train\events.out.tfevents.1584643236.JOHANN-
    LAPTOP.25888.3951176.v2")
731.        valFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs\\
    2020031901\\run2\\validation\events.out.tfevents.1584643237.JOHANN-
    LAPTOP.25888.3951198.v2")
732.        #run3
733.        trainFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs
    \\2020031901\\run2\\train\events.out.tfevents.1584644366.JOHANN-
    LAPTOP.25888.13123195.v2")
734.        valFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs\\
    2020031901\\run2\\validation\events.out.tfevents.1584644367.JOHANN-
    LAPTOP.25888.13123217.v2")
735.        #run4
736.        trainFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs
    \\2020031901\\run2\\train\events.out.tfevents.1584645958.JOHANN-
    LAPTOP.25888.28905284.v2")
737.        valFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs\\
    2020031901\\run2\\validation\events.out.tfevents.1584645959.JOHANN-
    LAPTOP.25888.28905306.v2")
738.        #run5
739.        trainFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs
    \\2020031901\\run2\\train\events.out.tfevents.1584646645.JOHANN-
    LAPTOP.25888.35118986.v2")
740.        valFile = os.path.join("D:\Johann\Workspace\Python\Tensorflow\Keras\mylogs\\
    2020031901\\run2\\validation\events.out.tfevents.1584646646.JOHANN-
    LAPTOP.25888.35119008.v2")
741.
742.
```

```
743.        Train_ea = event_accumulator.EventAccumulator(trainFile,
744.           ...:  size_guidance={ # see below regarding this argument
745.           ...:      event_accumulator.COMPRESSED_HISTOGRAMS: 500,
746.           ...:      event_accumulator.SCALARS: 0,
747.           ...:      event_accumulator.HISTOGRAMS: 1,
748.           ...:  })
749.
750.        Train_ea.Reload() # loads events from file
751.
752.
753.        Val_ea = event_accumulator.EventAccumulator(valFile,
754.           ...:  size_guidance={ # see below regarding this argument
755.           ...:      event_accumulator.COMPRESSED_HISTOGRAMS: 500,
756.           ...:      event_accumulator.SCALARS: 0,
757.           ...:      event_accumulator.HISTOGRAMS: 1,
758.           ...:  })
759.
760.        Val_ea.Reload() # loads events from file
761.
762.        #ea.Tags()
763.
764.        Train_ea.Scalars('epoch_accuracy')
765.        #Train_ea.Scalars('epoch_loss')
766.        #Val_ea.Scalars('epoch_accuracy')
767.        #Val_ea.Scalars('epoch_loss')
768.
769.
770.
771.        for T_acc,T_loss,V_acc,V_loss in zip(Train_ea.Scalars('epoch_accuracy'),Trai
    n_ea.Scalars('epoch_loss'),Val_ea.Scalars('epoch_accuracy'),Val_ea.Scalars('epoch_l
    oss')):
772.            print(str(T_acc[2]) + " : "+ str(T_loss[2]))
773.            Train_acc = np.append(Train_acc,T_acc[2])
774.            Train_loss = np.append(Train_loss,T_loss[2])
775.            Val_acc = np.append(Val_acc,V_acc[2])
776.            Val_loss = np.append(Val_loss,V_loss[2])
777.
778.
779.        fig = plt.figure(figsize=(19.2,9.49))
780.        plt.style.use('seaborn-whitegrid')
781.
782.
783.        major_ticks = np.arange(0, 101, 10)
784.        minor_ticks = np.arange(0, 101, 1)
785.
786.        ax = fig.add_subplot(1, 2, 1)
787.        ax.plot(Train_acc*100, color = 'C0',linestyle='solid',linewidth=1, label="Tr
    aining Accuracy")
788.        ax.plot(Val_acc*100, color = 'C1',linestyle='solid',linewidth=1, label="Vali
    dation Accuracy")
789.        title = 'VGG16 Training '
790.        ax.set_title(title)
791.        plt.legend(loc="upper left")
792.        ax.set_ylabel('Accuracy [%]')
793.        ax.set_xlabel('Epochs')
794.        #ax.set_xticks(major_ticks)
795.        #ax.set_xticks(minor_ticks, minor=False)
796.        ax.set_yticks(major_ticks)
797.        ax.set_yticks(minor_ticks, minor=True)
798.
799.        # And a corresponding grid
800.        ax.grid(which='both')
801.
802.        # Or if you want different settings for the grids:
803.        ax.grid(which='minor', alpha=0.2)
804.        ax.grid(which='major', alpha=0.5)
```

```
805.
806.
807.        major_ticks = np.arange(0, 11, 1)
808.        minor_ticks = np.arange(0, 11, 0.5)
809.
810.        ax = fig.add_subplot(1, 2, 2)
811.        ax.plot(Train_loss, color = 'C0',linestyle='solid',linewidth=1, label="Train
    ing Loss")
812.        ax.plot(Val_loss, color = 'C1',linestyle='solid',linewidth=1, label="Validat
    ion loss")
813.        title = 'VGG16 Loss '
814.        ax.set_title(title)
815.        plt.legend(loc="upper left")
816.        ax.set_ylabel('Loss')
817.        ax.set_xlabel('Epochs')
818.        #ax.set_xticks(major_ticks)
819.        #ax.set_xticks(minor_ticks, minor=True)
820.        ax.set_yticks(major_ticks)
821.        ax.set_yticks(minor_ticks, minor=True)
822.
823.        # And a corresponding grid
824.        ax.grid(which='both')
825.
826.        # Or if you want different settings for the grids:
827.        ax.grid(which='minor', alpha=0.2)
828.        ax.grid(which='major', alpha=0.5)
829.
830.
831.        plt.savefig('VGG16 Trainin_ACC_LOSS'+'.png', dpi=300,bbox_inches = 'tight',p
    ad_inches = 0.2)
832.        plt.close()
```