HAW HAMBURG

BACHELORTHESIS
Elias Soud

Classification of Electrical Resistors by a Deep Learning Model embedded in an Application for Mobile Devices

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Elias Soud

Classification of Electrical Resistors by a Deep Learning Model embedded in an Application for Mobile Devices

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr.-Ing. Marc Hensel
Second examiner: Prof. Dr. Annabella Rauscher-Scheibe

Day of delivery: 18. July 2022

Elias Soud

Title of Thesis

Classification of Electrical Resistors by a Deep Learning Model embedded in an Application for Mobile Devices

Keywords

Artificial Neurons, Artificial Neural Networks, Image Processing, Supervised learning, Reinforced Learning, Training Neural Networks, Parameters and Hyper-parameters, weight, bias, learning rate, batch size, epoch, data set, optimizer, Android.

Abstract

This work describes designing and training an Artificial Neural Network with Keras Tensorflow high level *Machine Learning* framework. this training enables the network to classify resistors within input images according to their value. The model is to be saved after training and evaluation, in order to then be employed in this project by an *Android* application.

Elias Soud

Thema der Arbeit

Klassifizierung von elektrischen Widerständen durch ein Deep-Learning-Modell, das in eine Anwendung für mobile Geräte eingebettet ist

Stichworte

Künstliche Neuronen, Künstliche Neuronale Netze, Bildverarbeitung, Überwachtes Lernen, Verstärktes Lernen, Training Neuronaler Netze, Parameter und Hyper-Parameter, Gewicht, Bias, Lernrate, Stapelgröße, Epoche, Datensatz, Optimierer, Android.

Kurzzusammenfassung

Diese Arbeit beschreibt den Entwurf und das Training eines künstlichen neuronalen Netzwerks mit Keras Tensorflow high level *Machine Learning* framework. Dieses Training

ermöglicht es dem Netzwerk, Widerstände in Eingabebildern nach ihrem Wert zu klassifizieren. Das Modell soll nach dem Training und der Auswertung gespeichert werden, um dann in diesem Projekt von einer *Android*-Anwendung verwendet zu werden.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Motivation . . . 1
1.2 Objectives . . . 1

2 Fundamentals 3

2.1 Background . . . 3
2.2 Computers . . . 4
2.3 Programming languages . . . 5
2.3.1 C++ . . . 6
2.3.2 Java . . . 7
2.3.3 Python . . . 8
2.3.4 Integrated Development Environment . . . 9
2.4 Operating Systems . . . 9
2.4.1 Linux . . . 10
2.4.2 Android . . . 10
2.5 Artificial Intelligence, Machine Learning, and Deep Learning . . . 11
2.6 Deep Learning . . . 12
2.6.1 Overview . . . 13
2.6.2 Artificial Neurons . . . 14
2.6.3 Loss Functions . . . 18
2.6.4 Gradient Descent, Epochs, and Learning Rate . . . 20
2.6.5 Batch Size and Stochastic Gradient Descent . . . 22
2.6.6 The training process of Deep Learning models . . . 24
2.6.7 Back-propagation . . . 25
2.6.8 Number of hidden layers and number of neurons . . . 27

2.6.9 Machine Vision and Convolutional Neural Network 27
2.6.10 Widely-used Deep Learning Libraries 32

3 Requirements Analysis 34
3.1 System context 34
3.2 Stakeholders 35
3.3 Overview of use case analysis 36
3.4 Functional and non-functional requirements overview 37
3.4.1 Functional requirements 37
3.4.2 Non-functional requirements 37
3.5 Defining the system's requirements 37

4 Concept and Design 39
4.1 Design flow 39
4.2 Development Environment 40
4.3 Programming languages, libraries, and frameworks 41
4.3.1 Python 42
4.3.2 C++ 44
4.3.3 Java 44
4.3.4 Alternatives 45
4.3.5 Dataset 45
4.3.6 Approach 46

5 Implementation 47
5.1 Training 47
5.1.1 Creation of the dataset 47
5.1.2 Model development 49
5.2 Integration into the Android application 55

6 Project results, insights, and conclusions 61
6.1 The training process 61
6.1.1 Problems 62
6.1.2 Insights 63
6.2 The Android application 64
6.2.1 Problems 64
6.2.2 Insights 65

6.3 Conclusions . . . 65
6.3.1 The deep learning model . . . 65
6.3.2 The Android application . . . 66
6.3.3 Working with images . . . 67
6.3.4 Final comments . . . 67

Bibliography **68**

A Appendix **72**
A.1 Back-propagation . . . 72
A.2 Trained models overview . . . 76

Declaration **103**

List of Figures

2.1 Programming languages popularity - image [9] 7
2.2 Comparison between the scope and approach of AI, ML, and DL. [4] . . . 11
2.3 Simplified comparison between classic programming and Machine Learning models from [23] 12
2.4 Simplified comparison between the traditional Machine Learning approach and Deep Learning. [15] 13
2.5 Performance of the top entrants to the ILSVRC by year. [15] 14
2.6 General structure of a perceptron. [7] 16
2.7 Sigmoid and tanh activation functions comparison. [13] 16
2.8 The Rectified Linear Unit activation function. [13] 17
2.9 Comparison of Adam to other optimization algorithms training a multi-layer perceptron. Taken from [14] 23
2.10 Example of a simple small-scale dense Artificial Neural Network. 24
2.11 Example of a model's cost function curve where there exists both a local and a global minima 25
2.12 Example of a convolutional layer kernel with an RGB image. [15] 29
2.13 Example of a convolutional layer's activation map with an RGB image. [15] 31
2.14 Example of a pooling layer. [15] 31

3.1 An outline of the system context 35
3.2 Use Case Diagram of the application's basic functionality 36

4.1 A visualized design of the application. 42
4.2 Some possible approaches for implementing the application. 46

5.1 Sample images of 150 Ω resistor from the produced dataset. 48
5.2 Code snippet for data handling and preparation in model-trainer.py script. 50
5.3 Code snippet for model fitting and callbacks in model-trainer.py script. . . 51
5.4 Code snippet for the model saving process in model-trainer.py script. . . . 51

5.5 Model 1 and 2 accuracy plot comparison 52
5.6 Model 3 and 7 accuracy plot comparison 52
5.7 Model 5 and 8 accuracy plot comparison 53
5.8 Model 9 accuracy and loss plots 53
5.9 Model 10, 11, 12, and 13 accuracy plot comparison 54
5.10 Code snippet for model access in CameraAccessActivity.java class. 56
5.11 Class diagram of the Android application with relevant methods. 57
5.12 Activity diagram of the Android application. 59
5.13 Snapshots of the application running on an Android device. 60

A.1 Model 1 summary 77
A.2 Model 1 accuracy and loss plots 78
A.3 Model 2 summary 79
A.4 Model 2 accuracy and loss plots 80
A.5 Model 3 summary 81
A.6 Model 3 accuracy and loss plots 82
A.7 Model 4 summary 83
A.8 Model 4 accuracy and loss plots 84
A.9 Model 5 summary 85
A.10 Model 5 accuracy and loss plots 86
A.11 Model 6 summary 87
A.12 Model 6 accuracy and loss plots 88
A.13 Model 7 summary 89
A.14 Model 7 accuracy and loss plots 90
A.15 Model 8 summary 91
A.16 Model 8 accuracy and loss plots 92
A.17 Model 9 summary 93
A.18 Model 9 accuracy and loss plots 94
A.19 Model 10 summary 95
A.20 Model 10 accuracy and loss plots 96
A.21 Model 11 summary 97
A.22 Model 11 accuracy and loss plots 98
A.23 Model 12 summary 99
A.24 Model 12 accuracy and loss plots 100
A.25 Model 13 summary 101
A.26 Model 13 accuracy and loss plots 102

List of Tables

2.1 Some of the well-known advantages and disadvantages of *C++* 8
2.2 Some ANN types and their corresponding well-suited applications 27

3.1 Stakeholder Analysis 35
3.2 Requirements analysis overview 38

A.1 Trained models 76

1 Introduction

1.1 Motivation

Modern advancements in technology, especially medical technology, have catapulted our capacity to further observe and understand how the human brain works. Of course, a lot still remains a mystery to be learned, but fortunately, thus far humanity was able to get to know some fundamental mechanics regarding biological vision and other sensory and non-sensory systems. [15]

Inspired by the knowledge of these biological systems and how they function, and enabled by the rise of efficient digital computing for researchers to use, computer scientists were able to somewhat digitally imitate these biological systems and construct artificial components such as Artificial Neural Network which are directly inspired by their physical counterpart, the neurons and neural networks in the brain.

Today, Artificial Intelligence, Machine Learning, and Deep Learning are well known terminology even in the mainstream public. This fame was earned by the increasing number of milestones and indeed impressive achievements in these domains. To add further, the ever-increasing number of applications that Machine Learning and Deep Learning models are able to tackle, from machine vision with object detection and recognition to autonomous vehicles, is raising interest in these topics even more.

1.2 Objectives

The goal of this work is the development of a software that employs a Deep Learning model or neural network to efficiently recognize and identify the value of a resistor within an image. The application runs on *Android* devices and also provides the ability to capture an image of a resistor to identify.

In Machine Learning, such application falls under the term “Classification Problem”, meaning, that the Deep Learning model is designed and trained to classify images into different “labels”, i.e. resistor values.

The network is designed and trained on a dataset using the Tensorflow and Keras frameworks in *Python* which is conveniently prepared for us in the *Google Colab* online python development environment. The program uses the *tensorflow C_API* to run the network model inside our app, and this *C/C++* code that runs the model is also wrapped with *Java* which is used of course to construct and implement our *Android* app.

2 Fundamentals

This chapter represents the essential theoretical foundation that is going to be relied on in order to implement the application. For the scope of this thesis, what is going to be covered in this chapter is only concerned with directly relevant topics and information such as Artificial Intelligence overview, some mathematics that are behind Machine Learning algorithms and mechanics, programming languages that can be used, and so on.

2.1 Background

The late 1970's into the early 1980's ushered in the beginning of what is now known as "The Digital Age" or "The Information Era" in which humanity still lives to this day. The rise of digital technology and the explosion of applications, both industrial and commercial, irrevocably transformed the lives of everyone. Digital communications technology, for example, made it so that two individuals are only a click of a button away from visually speaking to each other regardless of where these individuals are located, while the digitization of various industries made production easier, cheaper, and more efficient. One more further example of just how significant digital technology is of course the medical field which witnessed immense improvement and development from new machines that can provide medical personnel with essential new information and technical abilities that enable new forms of life-saving procedures and treatments.

One aspect or domain in this information era is what is now called "Artificial Intelligence"[1] or AI for short, and like many topics of its caliber it went through many different stages and phases. This topic has only really caught steam in late September 2012[2],

[1] Term was first coined in 1956 at a conference at Dartmouth College - New Hampshire - USA [17]

[2] September 2012 the ImageNet Large Scale Visual Recognition Challenge was impressively won by the AlexNet model [15]

as the impressive performance of the models involved has catapulted the topic into the mainstream, and it's been on the rise since then.

The fame that was commanded by AI models along with the implications and potential that was deduced from the AlexNet performance have lead to the explosion of interest in this domain in academia and even in private tech companies such as *Facebook*, *Google*, and *Amazon*. Of course, this interest have now been translated into solid achievements and milestones each year where it seems that AI is here to stay and it is going to be a big part of the future, even more than it already is now.

Until recently, all software was developed by programmers right from first principles. This meant that every single feature, mechanic, and behavior in any piece of software had to be addressed and manually [3] designed, coded, and implemented by programmers. At the present time this has changed with the emergence of more and more dependable AI models. In cases where the requirements are relatively simple and clear, where it is possible to achieve solutions that work 100% of the time rather cheaply and conveniently, then it is preferable, at this time, to not use AI based approaches. Other cases that involve some more complex problems is where AI approaches can really shine. [34]

2.2 Computers

This section touches briefly on computers in order to produce a simplified abstraction of what they are and how they function. Also in order to later understand the rational background of a programming language, it is helpful to have an understanding of how computers work.

According to the *Oxford Dictionary* [8], a computer is an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program. *Computers* carry out the different tasks they do by having specialized hardware electronic components that work together in order to execute those tasks. Those components are mainly *memory*, or ROM and RAM[4], *Central Processing Unit CPU* or *processor*, and storage which is typically an HDD[5] or SSD[6]. *Computers* usually

[3] I.e., a human was involved directly
[4] RAM or Random Access Memory is a relatively fast storage for computers and it stores variables and other data from the different programs running on the machine
[5] Hard-Disk Drive
[6] Solid-State Drive

have peripheral devices that enable input/output operations enabling human beings to experience and interact with those digital electronic machines.

Computers, being electrical machines, are powered by electricity. They use electricity and voltage levels to encode, store, and process information. Thus, *computers* encode voltage levels into zeros (low voltage) and ones (high voltage) and therefore the name of machine language is called "Binary".

2.3 Programming languages

Looking for a definition on *wikipedia.org*, the following results for a programming language can be found: "A programming language is any set of rules that converts strings, or graphical program elements in the case of visual programming languages, to various kinds of machine code output. Programming languages are one kind of computer language, and are used in computer programming to implement algorithms. Most programming languages consist of instructions for computers. There are programmable machines that use a set of specific instructions, rather than general programming languages." [32]

Programming languages are the means for human beings to communicate instructions and data to digital machines. They can also be understood as ways of encoding instructions and information in a way that can be understood by digital machines.

Programming languages vary in sophistication and human readability. Some languages like *Assembly* are considered to be low-level, meaning that they have less human readability because the structure and syntax reflect a close resemblance to how digital machines carry out instructions. It is accurate to say that the *Assembly* language is intended to communicate directly with a computer's hardware[7].

There are also higher-level languages, like *C++*, *Java*, *Javascript*, and *Python*. Those languages also have compilers or interpreters which translate their corresponding high-level language into machine code or platform-independent byte code. This goes on until all instructions and data are in binary form ready to be processed by the machine and influence its behavior. Each programming language has a wide range of APIs, libraries, and

[7] *Assembly* is closer to how machines communicate and carry out instructions, although computers can only understand and process instructions and various operations in binary form, as in zeros (0) and ones (1), and thus an *"Assembler"* translates that *Assembly* code into machine language

frameworks and in many applications programming languages work together to achieve the requirements of the given application.

Programming languages are used to construct and create all the digital software that most people use every day, they are also used to automate various tasks, and they vary in purposes and background and therefore they also vary in capabilities, with each language having advantages in certain applications while suffering from disadvantages in others. [18]

Figure 2.1 demonstrates how the popularity of different programming languages have evolved with time.

This section now lists the relevant programming languages for the development of this body of work. Each language's main characteristics are explored and a simplified comparison is outlined as well.

2.3.1 C++

C++ is a popular programming language that was originally designed by the Danish computer scientist *Bjarne Stroustrup* in 1983 at *Bell Labs* as an expansion of *C*. It is an extremely efficient[8], generic[9], object-oriented[10], and functional language.

Since it was made to be the child of the *C* programming language, it inherits from the *C* language the wide-ranging ability for low-level memory manipulation, which some consider to be one of the biggest advantages for *C/C++*. [24]

Like *C*, it is a compiled language, meaning it needs a compiler in order to be able to build *C++* applications and be able to use and execute them. There exists a wide range of *C++* compilers such as *Clang*, *GCC*, *Intel C++ compiler*, and *Microsoft Visual C++ compiler*, and these compilers can differ slightly but mostly the main structure remains intact. *C/C++* is also supported by a wide range of IDEs such as *Microsoft Visual Studio*, *Netbeans*, and *CodeBlocks*. *C++* also has a vibrant community behind it and an open-source base which is always helpful for individual programmers. [24]

Table 2.1 outlines some of the main characteristics of *C++*.

[8] Very fast when in execution time

[9] As in supports the kind of programming that allows for a later specification for different types in a given program

[10] I.e., it supports classes and objects

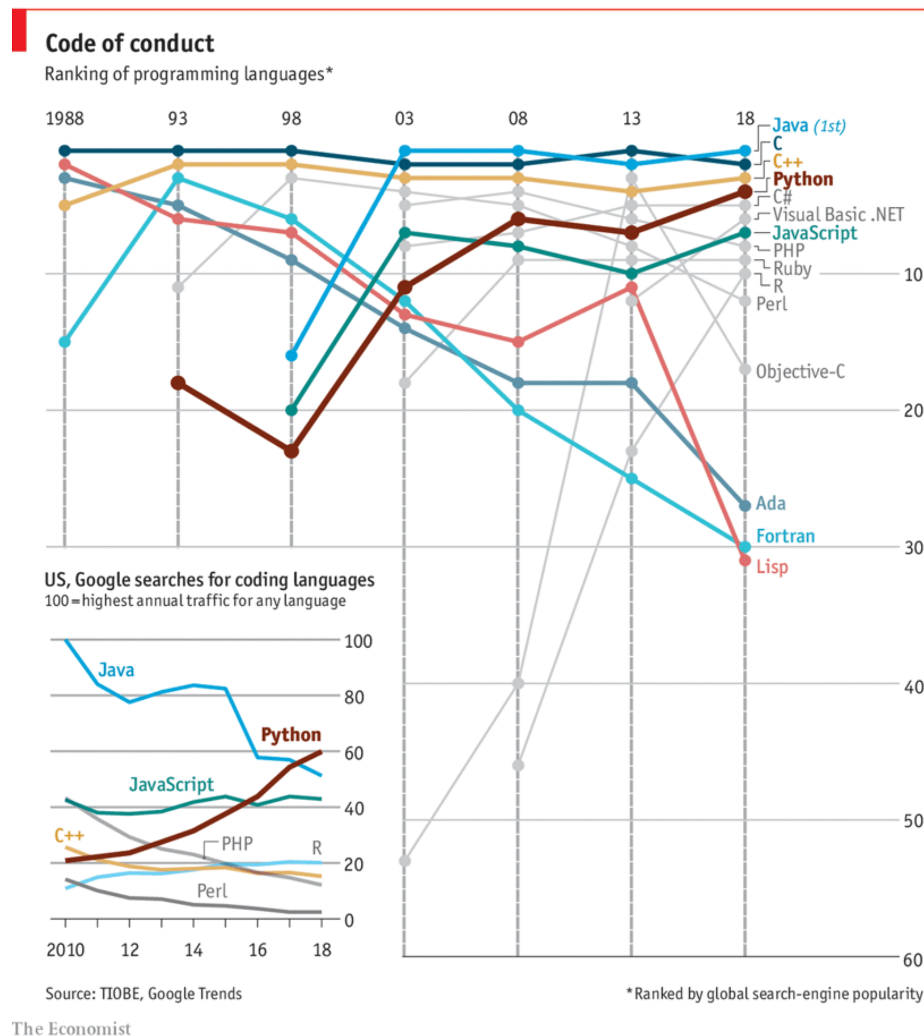


Figure 2.1: Programming languages popularity - image [9]

2.3.2 Java

Java is an object-oriented, class-based, high-level, programming language that requires checked and unchecked exceptions. Developed by *James Gosling* at *Sun Microsystems* and released in 1985. It is intended to implement the motto "write once, run anywhere", which means that compiled *Java* code can run on any platform that supports Java and without the need to recompile. *Java* applications are usually compiled into special *bytecode* that can run on any *Java virtual machine* regardless of the underlying hardware and architecture. The syntax of Java is similar to that of *C/C++*, but has fewer low-level

Advantages	Disadvantages
very high execution speed	relatively strict syntax
vibrant community with a lot of support and a huge range of open-source libraries	relatively difficult to learn
broader control over hardware resources	need for some experience in order to be able to harness the strengths of *C++*

Table 2.1: Some of the well-known advantages and disadvantages of *C++*

facilities than either of them. The official reference implementation of the *Java virtual machine* is the OpenJDK JVM which is free open-source software and used by most developers. *Java* is also supported by several IDEs like *Eclipse*. [18]

Java is one of the most popular languages for dynamic cross-platform deployment, and it was the official language for Android applications development until 2019 when it lost that position to Google's *Kotlin* language. However, *Java* can still be used for Android development today, and during the software development part of this thesis *Java* is used to design and build the Android application.

2.3.3 Python

Python is an object-oriented, interpreted[11] programming language with dynamic semantics[12]. It is a very high-level language and its built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development as well as for use as a scripting or glue language to connect existing components together. *Python's* simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. There is no compilation step, the edit-test-debug cycle is incredibly fast. The standard implementation of python is called "cpython" and it is the default and widely used implementation of *Python*. When a *Python* code is processed, it is converted into byte-code. An interpreter like *python virtual machine* takes care of the byte-code execution. [10, 12]

Python's incredibly high-level features and structure makes it an ideal language in academia and non-software-specific industry. Researchers utilize *Python* in an impressively broad

[11] Another interpreted language is Java, i.e. byte-code is "interpreted" into machine code

[12] Has an extremely dynamic and relatively non-strict typing as opposed to a language like C++ for example

range of domains and applications due to its easy-to-learn syntax. Its very high-level interfaces and libraries and tools allow the programmer using *Python* to focus more on their application and desired structure and outcome and less on the syntactical and non-syntactical restraints and strict development rules that come with other languages.

Python has no type declarations of variables, parameters, functions, or methods in source code. Instead, what *Python* does is that it tracks the types of all values at run-time and flags code that is invalid as it runs. [12]

2.3.4 Integrated Development Environment

An Integrated Development Environment, or IDE for short, is a kind of software program that combines many tools, frameworks, and functionalities that enable programmers to develop software with ease. Some of the tools and functionalities provided by IDEs include a code/text editor, debugging tools, compilation and built automation tools, and many others. IDEs hide many lower-level development steps and procedures, plus environment-related and compilation-related configurations, and combine everything in a more user-friendly Graphical User Interface which can drastically increase development productivity. [28]

There are many IDE variations but they mostly serve the same function, although there are some more usage-specific IDEs like *Code Composer* from *Texas Instruments* for embedded systems and microcontroller development or *Android Studio* by *Google* for *Android* development. The most widely-used general-purpose IDEs include *Microsoft Visual Studio*, *Eclipse*, *Codeblocks*, and *Netbeans*.

2.4 Operating Systems

Operating Systems are the link between the various computer programs and software on one hand and the machine's hardware resources on the other. *Operating Systems* manages memory and processes and they are considered to be what is called "system software"[13]. In most cases nowadays, there are several different programs running at the same time, and they all compete over the computer's hardware resources like the processor, memory, and storage. An *operating system* coordinates all of this to make

[13] *System software* is software designed to provide a platform for other software to run on.

sure that an efficient management is being carried out while several different programs and processes are running and demanding resources. [22, 25]

The most widely-used personal computer operating systems globally are *Microsoft Windows* with a market share of 76.45%, *macOS by Apple*, and all the different *Linux* distributions. Those operating systems are designed to be general-purpose personal usage systems but there are other embedded and dedicated systems that are specifically designed in order to achieve very specific requirements and tasks. Such systems can exist on micro-controllers or smaller light-weight computers[14].

2.4.1 Linux

Linux is a broad collection of open-source operating systems which are based on a Unix-variant called *Minix* or *Mini-Unix*. Since it is an open-source software, *Linux* has numerous distributions and variations but they are all built around the *Linux Kernel* which represents the core of the system and was originally designed by *Linus Torvalds* and released in 1991. *Linux* systems are widely used by programmers and computer scientists mainly due to their high performance and the amount of control *Linux* gives the user over the device's resources, but in recent years more and more distributions, like *Ubuntu* and *Fedora*, are becoming increasingly user-friendly. [22, 29]

2.4.2 Android

Android is an operating system that is dedicated primarily to touch-screen mobile devices, and it is based on a modified Linux kernel that comes with numerous open-source tools and other types of software. Core components are taken from the Android Open Source Project. *Android* is supported and sponsored by *Google* and the majority of *Android* devices run within the *Google* software base. It is written in *Java* for the UI, *C* for the core, in combination with *C++*. Android has been the best-selling OS worldwide on smartphone devices since 2011 and on tablet devices since 2013. [2]

[14] An example would be a *Raspbian Linux distribution* which can run on a *Raspberry Pi*.

2.5 Artificial Intelligence, Machine Learning, and Deep Learning

The terms *Artificial Intelligence, Machine Learning, and Deep Learning* are often used interchangeably, and while that was more acceptable a decade or two ago, now these terms have different meanings and definitions.

Figure 2.2 abstractly demonstrates the difference in scope between the three terms.

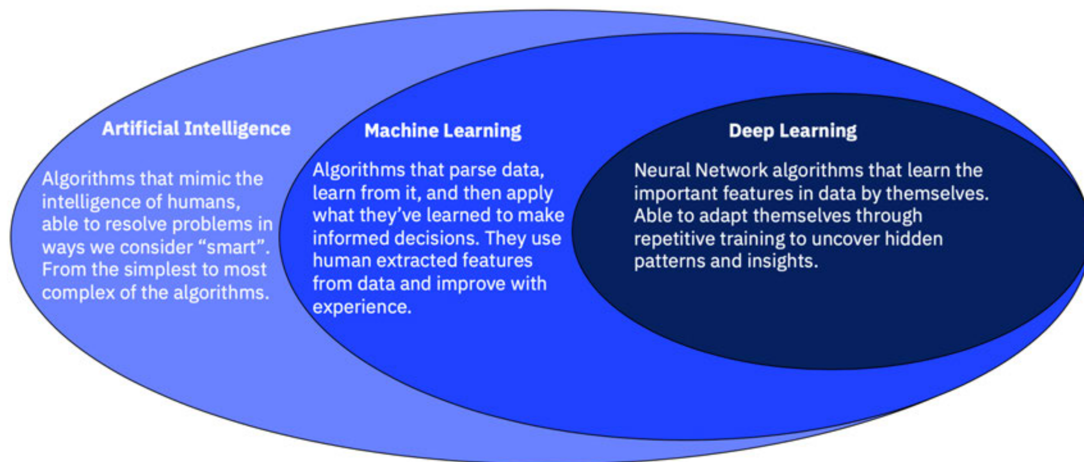


Figure 2.2: Comparison between the scope and approach of AI, ML, and DL. [4]

As was outlined in Section 2.1, software used to be designed and developed directly by programmers and professionals with utmost attention to the details and mechanics of every piece of functionality within the software. However, while the term Artificial Intelligence tends to refer to the general task of automating and imitating human-like behavior by "smart" machines, its subsets of Machine Learning and especially Deep Learning that are defined by more specific approaches to achieve and execute complex tasks have now changed the often restrictive approach of explicit programming in certain applications.

Figure 2.3 demonstrates a simplified visualization of the difference between traditional programming and machine learning. This comparison is concerned with the abstract inputs and outputs of each method.

To briefly touch on some of the distinctions between Machine Learning and Deep Learning, Machine Learning employs the statistical model and the algorithms that come with it in order to "learn" from previous experiences and while it has its own milestones and

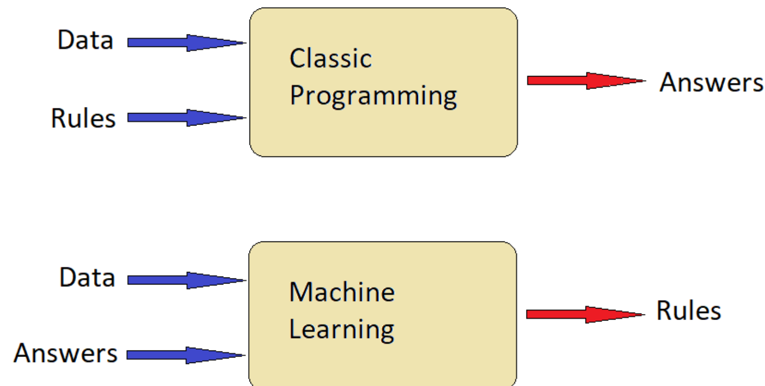


Figure 2.3: Simplified comparison between classic programming and Machine Learning models from [23]

achievements when it comes to how well it performs certain tasks, it involves procedures such as “feature engineering”[15] in addition to the relatively limited use of neural network models. [15, 23]

Deep Learning, which is the main focus for the scope of this thesis, extensively depends on ANN[16] and it involves significantly less, human-involved actions like feature engineering for the DL model to learn from input data. Deep Learning models excel at recognizing patterns in input data and DL models have recently risen to the top when it comes to performances when compared with its Machine Learning or even human counterparts.

Figure 2.4 abstractly illustrates the type of work that is involved in each domain. Deep learning involves more modelling while machine learning involves more feature engineering.

2.6 Deep Learning

Deep Learning is a subset of Machine Learning, it depends on the concept inspired by neural networks of the human brain which is known as *“Artificial Neural Network”* and

[15]The term *features* generally refers to the inputs of a certain ML model and feature engineering is the transformation of raw data into a well thought out and structured input variables.
[16]Artificial Neural Network, inspired by the brain’s neural networks.

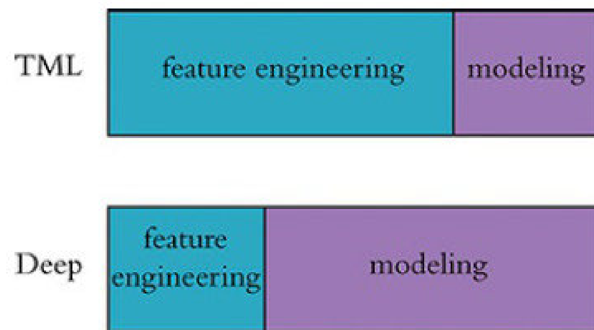


Figure 2.4: Simplified comparison between the traditional Machine Learning approach and Deep Learning. [15]

it uses a layered representation of data. Deep Learning uses multi-layered structures of algorithms and mathematical functions called *"Artificial Neurons"*[17]. [23]

Artificial Neural Network excel at detecting patterns in data, which means that a Deep Learning model can learn how carry out a task[18] with impressive above-human performance and without any explicit programming. [23, 34]

Figure 2.5 describes the performance of the top entrants to the ILSVRC or *ImageNet Large Scale Visual Recognition Challenge*[19].

The scope of this thesis will mostly focus on exploring a bit further how Deep Learning works in general terms, and how it can be employed in this thesis's Android application.

2.6.1 Overview

Every DL Network model has an input layer, an output layer, and at least one hidden[20] layer in between[21].

[17]Both inspired by the human brain, its neurons and neural connections and structures.
[18]Such as detecting and classifying objects in images, or simply learn how to play a video game.
[19]ILSVRC is an international competition for evaluating algorithms for object detection and image classification at large scale. [15]
[20]Hidden as is unexposed to the outside environment, like the case with the input and output layers, and these hidden layers are where the "learning" takes place.
[21]When a model has less than 3 hidden layers it would be called "shallow network".

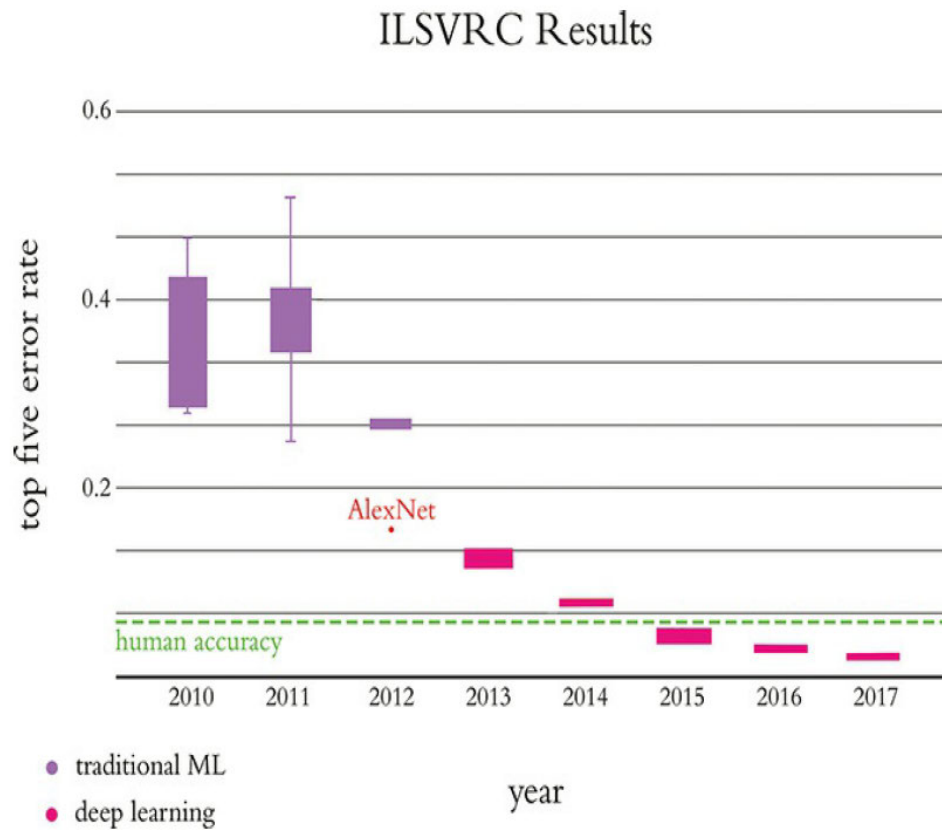


Figure 2.5: Performance of the top entrants to the ILSVRC by year. [15]

Hidden layers structure, type, and other characteristics can vary. This variation defines several kinds of hidden layers, among which are the *dense*[22], *convolutional*, or *recurrent* layers.

2.6.2 Artificial Neurons

Artificial Neurons are the building blocks of ANNs, and also are inspired by the neurons in the human brain. They are found in ANNs stacked in layers and connected according to a certain structure and their main purpose is to receive data from one or more inputs, operate on it by summing them, then finally producing a result as an output which is likely to be in itself an input to another neuron deeper in the network. In more technical

[22]They are the most general type of hidden layers. A layer is dense when each of its neurons are fully connected to all the neurons from the preceding layer.

terms, an Artificial Neuron is a mathematical function that takes a number of inputs which are then summed up to produce an output or *"activation"*[23]. [1, 15, 16, 34]

Equation 2.1 is the most fundamental equation in Deep Learning. It mathematically describes how artificial neurons calculate the weighted sum and adds the neuron's corresponding *"bias"* in order to later evaluate it through an *activation function*.

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{2.1}$$

In Equation 2.1 w_i represents the weights for each input and x_i represent the value of that input. Neuron bias is represented by b, while n is the total number of inputs for this particular neuron. [1, 15]

All these aforementioned parameters can have either positive or negative values, and in programming terms, they usually default to the float32 datatype not integers. [15]

The most basic kind of artificial neurons is what is known as *"Perceptron"* [21]. It is an algorithm for *supervised learning*[24] of linear binary classifiers. A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class [31]. The *perceptron* algorithm was invented in 1958 at the *Cornell Aeronautical Laboratory* by Frank Rosenblatt. The original paper is referenced in this work's bibliography [21]. The *perceptron* has a certain threshold and when the weighted sum is bigger than this threshold the *perceptron* will "fire" and the output would be equal to one, otherwise it will be equal to zero.

Fortunately nowadays, there exist some alternatives to the very functionally limited *perceptron* like the more practical and more capable *sigmoid* neuron which is based on a *sigmoid activation function* represented by Equation 2.2, or the more widely used *"Rectified Linear Unit"* or *ReLU* for short, or the *softmax* neuron which has a more specific activation.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

[23] Mathematical symbol for activation of a neuron is "a".

[24] Supervised learning (SL) is the machine learning task of learning a function that maps an input to an output based on example input-output pairs.[1] It infers a function from labeled training data consisting of a set of training examples. [33]

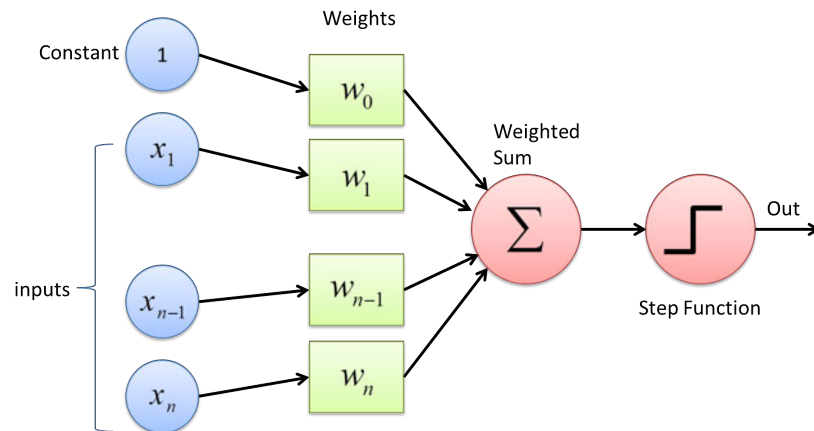


Figure 2.6: General structure of a perceptron. [7]

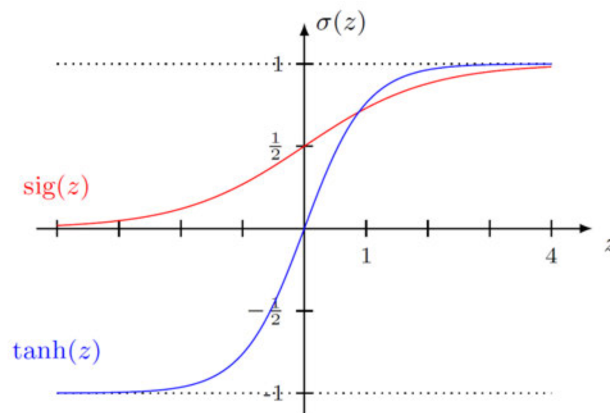


Figure 2.7: Sigmoid and tanh activation functions comparison. [13]

In Equation 2.2, $\sigma(z) = a$ where a is the neuron's activation. And z represents Equation 2.1, meaning $z = \sum_{i=1}^{n} w_i x_i + b$ as Figure 2.7 demonstrates.

The *sigmoid activation function* can represent gradual output values in contrast with the *perceptron* algorithm which only have two output states or values: "firing" or a 1, and "silent" or a 0.

A similar activation function to the *sigmoid* is the *tanh activation function*. The difference is that while the *sigmoid* range between 0 and 1 values, the *tanh* can have values from -1 to 1. Figure 2.7 outlines this difference between the two functions.

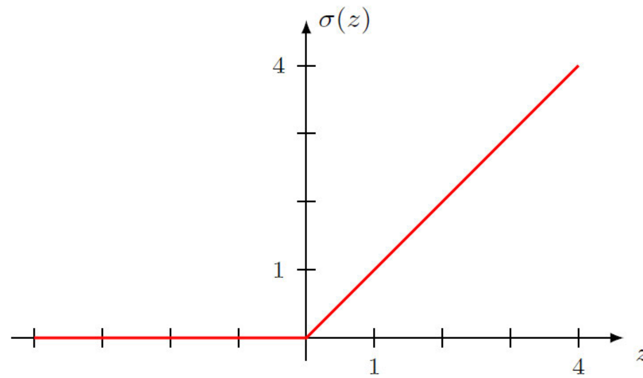


Figure 2.8: The Rectified Linear Unit activation function. [13]

The Equation that represents the *ReLU* function can be expressed in terms of given neuron's activation a and z which represents the given neuron's weighted sum from Equation 2.1:

$$a = \begin{cases} z: & \text{for z} > 0 \\ 0: & \text{for z} \leq 0 \end{cases} \tag{2.3}$$

Equation 2.3 can also be expressed as: $\text{a} = \max(0, \text{z})$. The *ReLU* neuron is the most widely used in Deep Learning applications due to several reasons, mainly its fast calculations characteristic in addition to decreasing the likelihood of *neural saturation*[25] from happening to the models neurons. It was also inspired by biological neurons and some of their properties

Each artificial neuron input has a corresponding weight which is multiplied by that input's value and then summed with other inputs of the same neuron, and the sum is then evaluated and compared according to a non-linear[26] function called *"activation function"*[27]. *Activation functions* are usually a non-linear mathematical function with some general characteristics such as gradually increasing, continuous, differentiable, and limited or bounded. Examples of such a function are the *sigmoid*, the *tanh*, and the *ReLU* functions. There are, of course, many other functions out there, some of which are derived from the *ReLU* like the *Exponential Linear Unit*. [1, 15, 34]

[25] A phenomena that limits the ability of neurons to learn after reaching a certain "saturation" level in its outputs, meaning it produces z values such that changes in z causes tiny negligible changes in the activation output a thus the learning capability of this neuron is drastically slowed

[26] Can be linear in certain applications such as regression.

[27] Also known as *"transfer function"*, but not to be confused with a linear system's transfer function.

When faced with a binary classification[28] problem then a *sigmoid* neuron would do the job in the network model's output layer. But if we had say a multi-class classification then usually a *softmax* neurons in output layer can suit such applications. [15]

Softmax neurons have been mentioned several times in this document, thus it can be helpful to understand how they work: Typically, *Softmax* neurons are used in output layers, and especially in multi-class classification problems. Based on its connected inputs, *Softmax* neuron will output the highest connected input value but it does so while providing the likelihoods of each class[29]. [15]

To summarize so far, for every artificial neural network model there are one input and one output layer and at least one hidden layer. The input layer only stores input values x in its neurons, the output layer represents the predicted output $\hat{y}$ of the entire given network model. The model's predicted output $\hat{y}$ is contrasted and compared with the true output value y of the corresponding input. Every connection between two neurons has a weight w and every neuron has a bias b.

This section will now explore topics that are relevant to the training and learning process of Deep Learning models.

2.6.3 Loss Functions

In order to qualify the spectrum of output evaluation and move closer and closer to the true desired output, i.e. train and learn, Deep Learning algorithms often involve what is called *"Cost Function"*, also known as *"Loss Function"*.

This section will touch some of the widely used *cost functions* categorized by the type of application. Both these categories that will be discussed are themselves a sub-category of *supervised learning* which is to be contrasted with the other type of learning in DL which is *unsupervised learning*.

[28] Binary classifications means the possible output is a binary, true or false, yes or no, and so on.
[29] Each *Softmax* neuron would represent a single classification class.

Regression problems

Regression problems revolve around predicting a continuous value[30]. Cost functions used in a regression problem are calculated on the distance-based error[31]. Examples from [15] of such functions:

- Mean Error (ME): The errors can be both negative and positive. So they can cancel each other out during summation and thus it is not a recommended cost function but it does lay the foundation for other cost functions of regression models.
- Mean Squared Error (MSE) also known as *"Quadratic Cost"*: This cost function solves the issue of *ME* above. Since errors are squared, it penalizes even small deviations in prediction $\hat{y}$ from the true value y. But if our dataset has outliers that contribute to larger prediction errors, then squaring this error further will magnify the error many times more and also lead to higher error. The *MSE* cost function is expressed in Equation 2.4.

$$MSE = \frac{1}{n} \cdot \sum_{i=1}^{n} (y - \hat{y})^2 \tag{2.4}$$

- Mean Absolute Error (MAE) in contrast with *MSE* it is robust to outliers thus it will perform better even when dealing with a dataset that has noise or outliers, expressed in Equation 2.5.

$$MAE = \frac{1}{n} \cdot \sum_{i=1}^{n} |y - \hat{y}|^2 \tag{2.5}$$

Classification problems

When dealing with a problem where the output variable is a category, such as *true* or *false*, *yes* or *no*, *blue* or *red* or *orange* or *green*, we call such problems *Classification problems*, and there are specific *cost functions* that are well suited to such applications. [1, 15]

One of the *cost functions* which are typically used for classification problems is the *Cross-Entropy Cost function* expressed in Equation 2.6. This function is one way to limit and

[30]Examples of a continuous value: salary of an employee, price of a house or a car, temperature, etc...
[31]Error $= y - \hat{y}$: where y is the true output and $\hat{y}$ is the predicted one.

minimize the impact of neuron saturation on learning speed, due to it being configured to enable efficient learning anywhere within the activation function curve.

The *Cross-Entropy Cost function* is structured in a way so that the derivative that represents the rate of change of the *cost* C with regard to the trainable parameters is related to the term $(y - \hat{y})$, and thus the larger the difference between ideal output y and estimated output $\hat{y}$[32] the greater the rate of change of the cost with regards to the parameters. [15]

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln \hat{y}_i + (1 - y_i) \ln (1 - \hat{y}_i)] \tag{2.6}$$

The C in Equation 2.6 refers to the *cost*.

2.6.4 Gradient Descent, Epochs, and Learning Rate

This section will dive deeper in the relevant conceptual topics and terms in Deep Learning and will explore further details in the learning and training process.

Gradient Descent

Gradient Descent is an effective computational method or algorithm that can be used for adjusting a DL model's trainable parameters with the aim of minimizing the *cost*. It is particularly useful when there is a lot of training data available[33].

It is widely used in Machine Learning and Deep Learning domains. It fundamental idea is adjusting the parameters of a model, seeking to reach the lowest possible cost. The amount of the adjustment is called *"step size"* and is described by Equation 2.7 where η represents the learning rate. [15]

$$stepsize = \eta \cdot \frac{\partial C_w}{\partial w} \tag{2.7}$$

[32] When there is a single output neuron in the output layer, it can be stated that the predicted output of the network model $\hat{y}$ is the activation a of that output neuron.

[33] A good general rule in ML and DL domains is: the more the data, the better.

Learning Rate η

Learning rate describes the rate in which the trainable parameters are modified, i.e. it influences the *"step size"* of the gradient descent or the amount of which it changes value. [15]

It is considered one of several DL model's *Hyper-parameters*[34], other *Hyper-parameters* include *batch size* and *hidden layers count*.

When the *Learning rate* is too small then it would take too many iterations of gradient descent in order to reach the minimal cost. When it is too big it can mean that the gradient may never reach the minimal cost. Therefore choosing a moderate initial value for it, like $\eta = 0.01$ or $\eta = 0.001$, and observe if the model learns[35] well or not. If the cost decrease was small with each iteration then increasing the *learning rate* by an order of magnitude can help improve this, but if the cost jumps up and down then it is possible that the *learning rate* is too high and it is possible that it should be decreased by an order of magnitude. [15]

While the *learning rate* is recommended to have a moderate initial value, it is worth mentioning that it is also recommended that the weights be initialized with random values according to a *distribution* like the *Xavier Glorot distribution*.

This prevents the initial neurons activation from having extreme values[36] and instead it makes it so that the model activations have a typically ideal distribution which is a *normal distribution* with a mean of nearly 0.5. The biases are typically initialized with a zero.

At this point it is helpful to get familiar with another term, which is *"Epochs"*. An *epoch* is basically a training iteration or a complete training cycle. An *epoch* represents one complete training cycle in which each sample in the training dataset has had a contribution to update the internal model parameters. The number of epochs is also a *hyper-parameter* that defines the number times that the learning algorithm will work through the entire training dataset before it finishes training. And like many of a model's hyper-parameters,

---

[34]Hyper-parameters are variables of a DL model that can be modified and configured before the start of the training process.

[35]I.e., if the model's cost decreases consistently with each training iteration.

[36]Initially when starting the training process, initializing weights with extreme values means that the model would have many unmerited strong opinions about the input and that can lean to wide-spread neuron saturation and other learning obstacles and hindrances.

choosing the optimal value for a particular application takes considerable amounts of trial and error and observing the impact every change has on the model's cost.

There are *"early-stopping callback functions"* which are methods that automatically monitor training and validation costs and stop the training early, i.e. before the total number of epochs is reached, if they detect a pattern of increasing costs. This behavior allows setting a relatively arbitrarily-high number of training epochs while avoiding *overfitting*. Keep into consideration that *overfitting* describes a training consequence that takes place when a model trains for so long to a point where it becomes too familiar and too adapted to the training data, that it loses its ability to generalize predictions over unseen data[37]. In more technical terms, it is when the training cost continues to decrease while the validation cost increases. *Overfitting* may also indicate that a given model is too complex for the task at hand and should be simplified. [15]

There are numerous ways that help models to avoid *overfitting* a particular training dataset. These methodologies include for example:

- Simply increasing the amount of training data or size of the dataset.
- Reducing the number of training epochs.
- Using an *early-stopping callback function*.
- Implementing *dropout*.
- Implementing *L1/L2 regularization*.
- Implementing *batch normalization*.

Underfitting is also a problem and it takes place when a model has to deal with too much complexity while having too few parameters to train. [1, 5, 15, 23, 34]

2.6.5 Batch Size and Stochastic Gradient Descent

When dealing with a large quantity of training data, ordinary gradient descent may not work because it isn't possible to fit all of the data elements in the memory RAM. And when dealing with models with too many trainable parameters[38], regular gradient

[37] Like the data that weren't included or available in the training dataset during the training process.
[38] Again, these parameters are the weights of connections in the model and the bias of each neuron

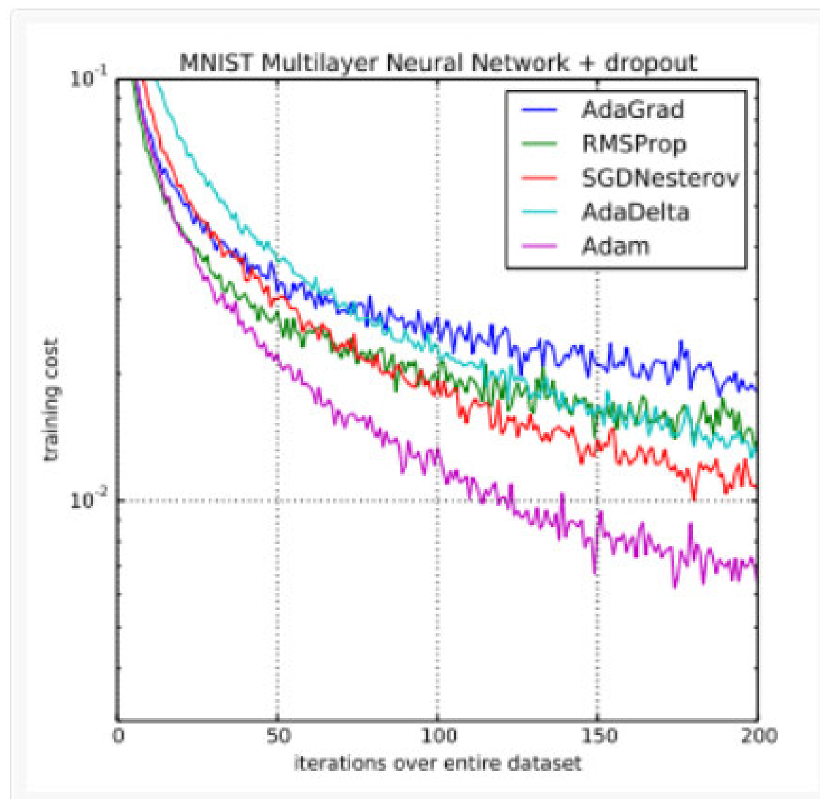


Figure 2.9: Comparison of Adam to other optimization algorithms training a multi-layer perceptron. Taken from [14]

descent would also likely be impractical and ineffective because that would mean a lot of processing power is required for training. [15]

A solution for these limitations is called *Stochastic Gradient Descent* or SGD for short. *Stochastic Gradient Descent* is what is categorized as an *optimizer*[39]. It is a variation of regular gradient descent. In this variation the training dataset is split into *mini-batches*[40] to make gradient descent manageable and productive. [15]

The *batch size* is another hyper-parameter that represents the size of the aforementioned mini-batches. It describes the number of training data points used for a given iteration

[39]Other popular optimizer algorithms include *momentum*, *AdaGrad*, *AdaDelta*, *RMSprop*, and *Adam* which is short for *adaptive momentum*. Each algorithm has advantages and disadvantages but *Adam* is a great deep learning optimizer that is well suited to noisy problems as it adjusts parameters separately. *Adam* was presented by *Diederik Kingma* from *OpenAI* and *Jimmy Ba* from the University of Toronto in their 2015 ICLR paper titled “Adam: A Method for Stochastic Optimization”.

[40]Each mini-batch is a subset of the dataset with a number of mini-batches is equal to the total number of data points in the dataset divided by the batch size.

of the SGD and it directly affects the amount of memory usage of the training process. [15]

2.6.6 The training process of Deep Learning models

To recap the knowledge base discussed so far in this document, by now it is established that Artificial Neural Networks are made up of layers, each layer consists of artificial neurons of a certain type each with weighted connections and a corresponding bias, the hidden layers neurons receive weighted inputs which are summed as expressed in Equation 2.1. Thus, input data forward propagates through the model where a prediction will be made in the output layer. That prediction is then compared to the true desired value of that corresponding input sequence, thus producing a *cost* value that is desired to be minimized and decreased by each passing *epoch*. Figure 2.10 demonstrates a simple example of a shallow dense ANN.

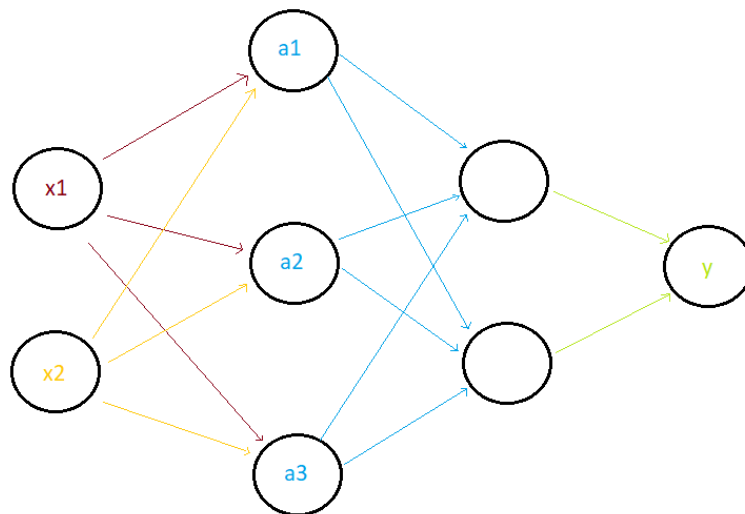


Figure 2.10: Example of a simple small-scale dense Artificial Neural Network.

Local minima and global minima in cost function curve

When the gradient descent "descents" over the cost curve, it is seeking the point in the curve, called *global minimum*, where the cost is at its lowest possible value with regards to the entire curve. Gradients could get stuck in what is known as a *local minimum*, a point where the cost is the lowest in a given segment of the cost curve but it is not the lowest in the entire cost curve. This problem is touched on graphically in Figure 2.11.

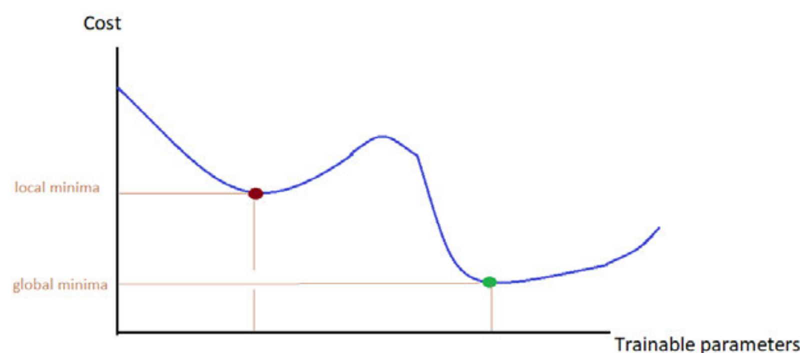


Figure 2.11: Example of a model's cost function curve where there exists both a local and a global minima

Stochastic Gradient Descent or SGD helps with avoiding falling and getting stuck in a local minima due to its randomized and shuffled batching of the training dataset. Other methodologies to reduce the chance of falling into local minima are explored a bit further in later sections of this document based on their relevance to the implementation of this work.

For example, increasing the gradient descent's step size, i.e. *learning rate* η, can also help in avoiding local minima. Another example is choosing a good batch-size[41], as it is typically recommended not to go above a batch-size of 128 to decrease the chances of being trapped in a local minima.

2.6.7 Back-propagation

SGD on its own is practically good enough to adjust a model's parameters and minimize the cost in many types of ML models, but for DL models it is necessary to efficiently

[41]A typically good initial value for batch-size is 32

adjust parameters through multiple layers of neurons, each layer having a different scale of impact to the model cost. Thus DL typically partners with a technique called *"Back-propagation"* that can be contrasted with *forward-propagation* which is the forward[42] traversal of data through the model's layers in order to finally produce the model's prediction or output. [15]

Back-propagation carries information about the cost C backwards in reverse order, i.e. from the output layer where the cost is calculated and then going backwards through the model's layers, with the aim of reducing cost by adjusting neuron parameters accordingly throughout the network model. [15]

The mathematics behind back-propagation involve partial derivative calculus. This document's Appendix A has a back-propagation section that touches on some of that math behind it. Keep in mind that in back-propagation, the amount by which each weight across the network is adjusted is proportional to the cost function's gradient with regards to that particular weight, and this adjustment is made in the aim of reducing the cost. Back-propagation utilizes the cost to calculate the relative contribution of every single parameter to the total cost which is represented by the cost gradient with regards to that parameter. Then, it updates each of those parameters accordingly, increasing or decreasing it depending on which is associated with a reduction of the cost, by an amount equal to the step size from Equation 2.7. And that is how a DL network model, in an iterative manner, reduces cost, and learns.

A fair deduction from what has been discussed so far is that the gradients that are used to adjust the weights of a layer that is far away back from the output layer would be very small. Thus the effects of back-propagation could somewhat vanishes in the early layers far away from the source of the cost and back-propagation which is the output layer. This phenomena is called *vanishing gradients*, and it is one of the problems and obstacles that can be addressed in DL applications.

With regards to back-propagation, it can be summarized with:

1. Finding the error of the cost function δ_L
2. Use δ_L for the calculation of the derivative of the cost function with regards to the weights in layer L

[42]Forward here means starting at the input layer and ending up with the output layer

3. Finding δ_{L-1} then using it to calculate the gradient of the cost function with regards to the weights in layer $L-1$

4. Repeating the process for every layer until reaching the *first hidden layer* which is the last layer with trainable parameters

2.6.8 Number of hidden layers and number of neurons

Having more hidden layers in a given DL model allows for deeper and more complex capacity for abstraction but it also can cause problems like *vanishing gradients* where back-propagation becomes less effective the farther away from the output layer it goes. This can be solved by several means but that in turn involves some trade-offs like requiring more hardware resources and processing power. [15]

A larger number neurons in a network model can lead to more complex learning and pattern recognition but also would require more processing power and would take longer to train while having too few neurons can lead to a decrease in the model's accuracy and performance. [15]

2.6.9 Machine Vision and Convolutional Neural Network

Definitions of the term *machine vision* vary, but all include the technology and methods used to extract information from a digital image in an autonomous[43] manner. [30]

In general and in many cases, it can be stated that different network architecture and structures perform differently in various types of applications. Table 2.2 outlines, in general terms, each network type and the application in which it performs best.

Type of network	Best-suited type of applications
Recurrent neural networks RNN	Natural language processing
Generative adversarial networks GAN	Visual creativity
Convolutional neural networks CNN	Machine vision

Table 2.2: Some ANN types and their corresponding well-suited applications

[43] Autonomous here means certain types of software that are able to process and handle digital images and extract information from them.

Convolutional Nerual Networks (CNN)

It can be said that a DL network model is *convolutional* when it consists of at least one *convolutional* hidden layer. *Convolutional* layer types enable a DL model to efficiently process spatial patterns. [15]

Visual images are structured as a two-dimensional arrays of pixels. For a colored image, there would typically be 3 channels, e.g; *RGB*[44], for expressing every pixel. So there would be three two-dimensional matrices, each represents a corresponding color for the image pixels. [15]

Dense network models can handle an image as an input, only it is required to *flatten*[45] the image pixel data first so that it is possible to assign each pixel in the flattened image vector to an input neuron. One major problem with *flattening* an image into a vector is that the data loses significant information about the structure of the visual image and its spacial patterns. [15]

Inspired by biological vision, CNNs are a form of ANN adaptation to be able to better deal with visual imagery.

Convolutional layers

Convolutional layers consist of sets of *kernels* or *filters*. Every kernel is a sub-window that scans across an image from top left to bottom right, and it is made up of weights which, as in dense layers, are learned through back-propagation. *Kernels* typically vary in size but common sizes are (3×3) and (5×5) and thus a (3×3) kernel for an RGB image would have $(3 \times 3 \times 3)$ weights. [15]

As demonstrated in Figure 2.12, kernel weights do not change value as the kernel moves across an image but instead they are shared across all the inputs which are an image sub-window of pixels. This means that convolutional layers have orders of magnitude less weights than a dense layer. The output of a kernel is made up of all kernel activations[46], and it is arranged as a 2-D array. [15]

[44] RGB signify the three colors Red, Green, and Blue, each one having its dedicated channel for each pixel.
[45] A 2-D matrix can be "flattened" into a 1-D matrix, which is a vector
[46] And that is why a kernel output is called an "Activation Map"

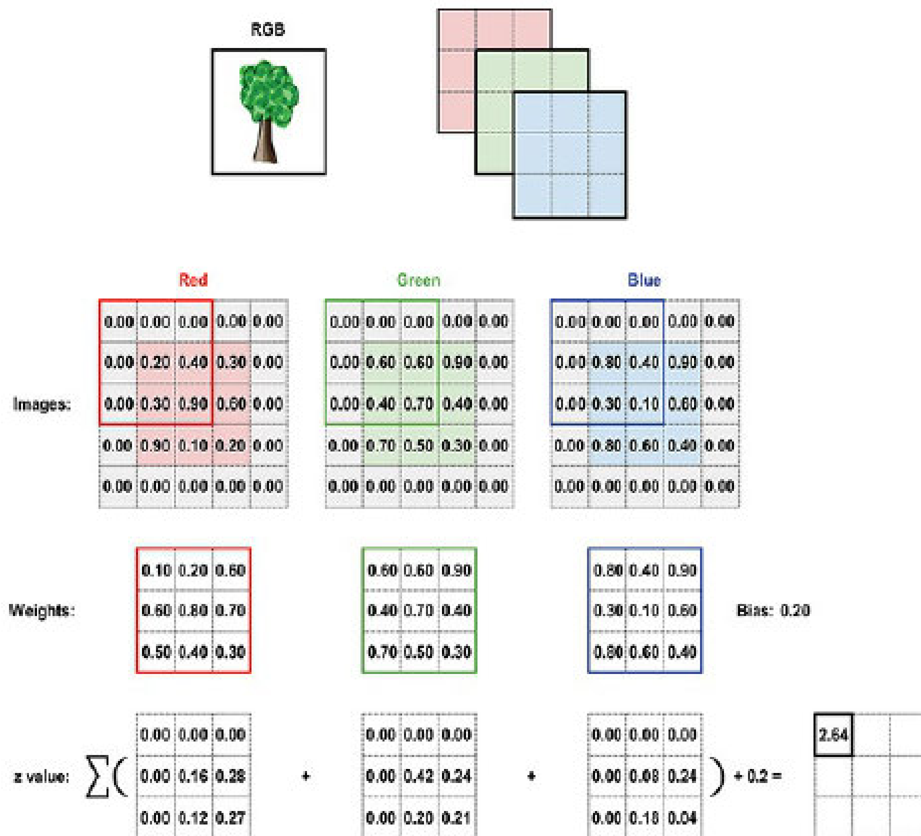


Figure 2.12: Example of a convolutional layer kernel with an RGB image. [15]

Typically, there would be multiple kernels for each conv-layer, each allowing the network to learn a representation of the data in a unique way. For example, a filter that responds optimally to vertical lines or color transitions would produce a large activation value a when it detects those features in the sub-window of the image. [15]

As the network model gets deeper having more layers, those filters in the convolutional layers react to more complex combinations of the preceding layer's simpler features, learning to represent increasingly abstract spacial and color patterns until the final layers have the ability to recognize entire objects.

The number of filters in each convolutional layer, similar to the number of neurons in a dense layer, is a model hyper-parameter. Other CNN hyper-parameters are:

1. Kernel size: a common size is 3×3.

2. Stride length or step length: this hyper-parameter refers to the size of the step that the kernel takes as it moves across an image.
3. Padding: along with the stride, it plays a role keeping the ratio between the image size, the filter size, and the step size so that the filter doesn't *"overflow"* over the image edge and thus padding adds neutral (or zero) values. This is expressed in equations 2.8 and 2.9 where the activation map needs to be a valid value, i.e. an even integer.

$$W_a = \frac{W_i - F + 2P}{S} + 1 \tag{2.8}$$

where:
W_a : width of activation map
W_i : width dimension of an image
F : size of kernel
P : amount of padding
S : stride length

$$H_a = \frac{H_i - F + 2P}{S} + 1 \tag{2.9}$$

where:
H_a : height of activation map
H_i : height dimension of an image
F : size of kernel
P : amount of padding
S : stride length

Pooling Layers

A convolutional layer can have any number of kernels, each producing an activation map and thus the output of a convolutional layer is a 3-D array of activation maps with the depth reflecting the number of kernels in the layer. [15]

Pooling layers work hand in hand with convolutional layer, helping to reduce the overall count of parameters as well as complexity, thereby speeding up computation and helping with avoiding overfitting. [15]

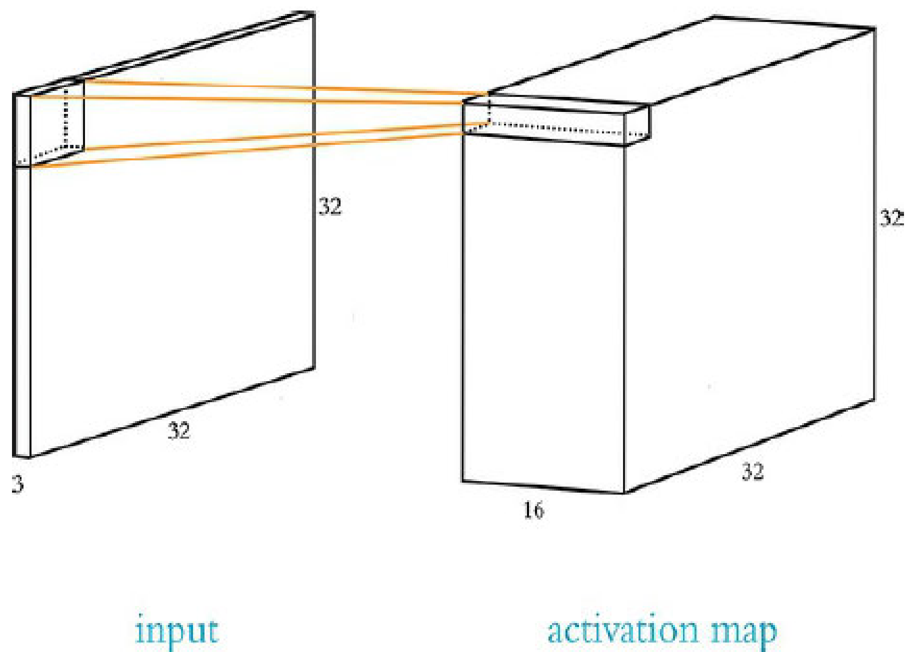


Figure 2.13: Example of a convolutional layer's activation map with an RGB image. [15]

Pooling layers also have a stride length and filter size, and these filters also slide over its input, applying data-reduction operations.

Pooling layers most often use the *max* operation, retaining the largest value in its sub-window as in Figure 2.14 where the stride is 2×2 for horizontal and vertical strides. There is also *average pooling* and other types of pooling.

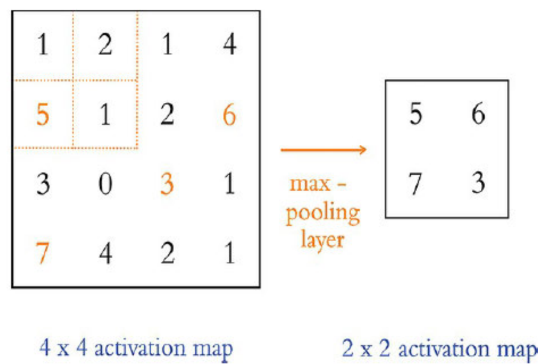


Figure 2.14: Example of a pooling layer. [15]

Back-propagation and CNNs

During training, CNNs keep track of the index of the max value in each forward pass, such that the gradient for that particular weight is back-propagated properly and is used to update and modify the correct parameters. It is common to have a group of two or three consecutive convolutional layers followed by a pooling layer, all culminating into *conv-pool* blocks which can be repeated several times. It is also typical to have these *conv-pool* blocks culminate into a dense hidden layer, or a several of them, and ends with the output layer. [15]

Object Detection

In Deep Learning general terms, *object detection* aims at finding where objects are. Contrast that with *object classification* which aims at finding what are the detected objects and classify them. The list for the famous *object detection and classification* models include *R-CNN* [11], *faster R-CNN* [20], and *YOLO* [19].

2.6.10 Widely-used Deep Learning Libraries

Some of the most widely-used frameworks and libraries for deep learning are:

1. *Tensorflow*[47]: best-known and most widely-used, developed by *Google* who later made it open-source.
2. *Keras*[48]: uses *Tensorflow* in the background and thus focusing on more user-friendly development.
3. *pyTorch*[49]: developed by *Facebook* and is strictly limited to *python*.
4. *MxNet*[50]: developed by *Amazon*.
5. *CNTK*[51]: *Microsoft's* cognitive toolkit.

[47] https://www.tensorflow.org/
[48] https://keras.io/
[49] https://pytorch.org/
[50] https://mxnet.apache.org
[51] https://docs.microsoft.com/en-us/cognitive-toolkit/

As always, there is a multitude of other topics to talk about when discussing Deep Learning such as *Residual Networks and Residual connections* and *Transfer Learning* but for the scope of this document, only networks that are relevant to image classification are explored.

3 Requirements Analysis

This chapter is concerned with the requirement analysis of the application. A definition for requirement analysis can be taken directly from *wikipedia.org*[1]: "In systems/software engineering, requirement analysis focuses on the tasks that determine the needs or conditions to meet the new or altered product or project". In other words, it is a way for engineers to develop systems/software in accordance with specific standards, criterion, and functionality in a concise informative structure [26]. It is a process used to determine the needs that a system should meet to achieve a particular task. Requirements also vary in type, such as functional requirements and non-functional requirements, and they include what is known as stakeholder requirements in addition to use case diagrams[16].

3.1 System context

A system context describes the environmental factors that may affect the performance and behavior of a system. Figure 3.1 clearly defines lighting conditions and a resistor's distance from the camera as environmental factors that could have an influence over the performance of the system.

Defining a system context is also helpful for the development process, as it gives the developer the ability to carry out the construction of a system while taking into consideration the possible hurdles or obstructions that a given system could encounter when put in practical use.

[1] https://en.wikipedia.org/wiki/Requirements_analysis - date: 20/May/2022

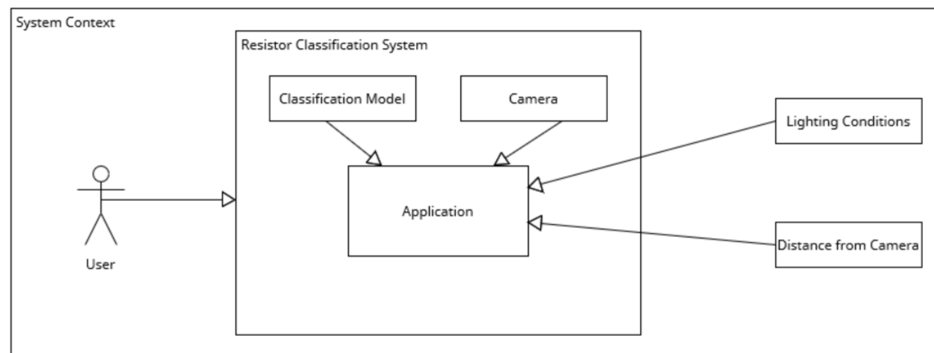


Figure 3.1: An outline of the system context

3.2 Stakeholders

When designing software, identifying the stakeholders and their specific requirements, that are involved in the behaviour of the application. It is an essential part of the requirement analysis phase of development [26].

This stakeholder analysis defines the possible users who have a vested interest in the system's development and those who directly interact with the application [26].

Taking into consideration the significance of each requirement to the functionality of the application, a list of the system's stakeholders is formalized in Table 3.1.

Stakeholder	Interest
Users	The system can be used for research purposes and it can be functionally expandable, improved, and built upon. Also a reliable system performance that enables users to classify resistors efficiently
The Developer	The creator of this software has a direct interest in the applications success, efficiency, portability, maintainability and ability to be upgraded
The Supervising Professors	The supervising examiner Prof. Dr. Hensel has a direct interest in AI and its applications on Android, in addition to the re-usability of the software components. The supervisor also has an interest in acquiring a well-documented foundation for future thesis

Table 3.1: Stakeholder Analysis

3.3 Overview of use case analysis

Use case analysis is a technique used to identify the functional requirements of a system and to define the ways in which an system is used. The use case analysis is the foundation upon which the system will be built and the primary form for gathering usage requirements for a new software program or task to be completed. The primary goals of a use case analysis are: designing a system from the user's perspective, communicating system behavior in the user's terms, and specifying all externally visible behaviors.[2]

In this section, a clear analysis of the usage of the application will be carried out. Chapter 4 lays out in more detail the methodology behind implementing the design which achieves the use-cases and other requirements outlined in this chapter.

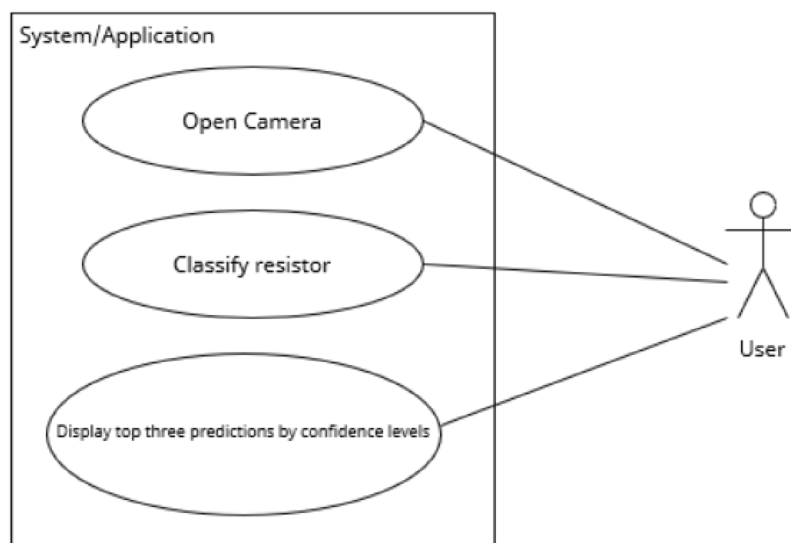


Figure 3.2: Use Case Diagram of the application's basic functionality

As shown in Figure 3.2, From the user's perspective, the only interactions/use-cases that a user of the application can carry out are firstly initiating an image capture by clicking the dedicated button in the UI and secondly reading the resulting evaluation of the image by the application via the dedicated label display area in the UI. The resulting use case analysis is relatively simple as the only user-level functionality is taking a picture of a resistor to predict its value and display it on the UI.

[2]https://en.wikipedia.org/wiki/Use-case_analysis - visited on: 20/May/2022

3.4 Functional and non-functional requirements overview

In software systems engineering, it can be helpful to categorize different requirements based on some criteria in order to acquire a more detailed understanding of our system and how to go about developing and implementing it. The two main types of requirements are functional and non-functional requirements, and both are utilized as categories here for the requirements analysis of the system.

Functional requirements define particular functionality of the system, While non-functional requirements define the overall characteristics such as cost and reliability[26].

This section describes those distinctions and provides the requirement analysis of the overall system of this thesis.

3.4.1 Functional requirements

Functional requirements are tightly connected with the use cases of the system. Functional requirements defines a function of a system, where a function is described as a specified behavior between inputs and outputs. Functional requirements may involve calculations, data manipulation[3].

3.4.2 Non-functional requirements

Non-functional requirements are another component of requirement analysis. It specifies the metrics that can be used to evaluate the operation of a system[4].

3.5 Defining the system's requirements

This section employs the definitions from Section 3.4 for the purpose of concretely defining the systems requirements. Table 3.2 serves exactly this purpose as it culminates the main requirements of the application into the two requirements categories from Section 3.4.

[3]https://en.wikipedia.org/wiki/Functional_requirement - date: 22/May/2022
[4]https://en.wikipedia.org/wiki/Non-functional_requirement - date: 22/May/2022

Functional requirements	Non-functional requirements
System must be able to produce an image frame from camera feed	System must be able to display the top three results within two seconds
System must be able to detect a single E24 series resistor located in the camera frames	System must be easily scalable and maintainable
System must be able to classify detected resistors into their corresponding value	System must run on an *Android* device with an API level of 25 or above to balance between new development features and compatibility with older devices
System must be able to display a detected resistor's predicted value along with the certainty of the prediction on the UI for the user to read	System must maintain a minimum prediction accuracy of 90%
System must have a UI that provides the user with a brief usage guide and a button to initiate camera feed and evaluation process and displaying evaluation results on UI	System must be able to process the produces frames before the evaluation process

Table 3.2: Requirements analysis overview

4 Concept and Design

This chapter outlines the conceptual blueprint for implementing the system as well as utilizing the theoretical foundation that was researched and collected in Chapter 2 in order to put together a conceptual design for the system. A design that would enable it to achieve the requirements laid out in Chapter 3.

4.1 Design flow

This section demonstrates an overview of the general flow of the design that is being implemented and some of the reasoning behind it.

- The application runs on an Android device with an API level of 25 or above. The development process is carried out by using the *Java* programming language with the *Android Studio* IDE for constructing the UI, binding events to actions, and handling the communication with the Android operating system and its hardware.
- The UI contains some usage instructions for the application in text form in addition to an interactive button that will trigger a request to start the camera. Then, the application will be able to access the *Android Camera Interface* and relay the frames to the application's Native[1] library.
- The native library of the application processes the incoming frames received from the Java wrapper into RGB bitmaps so the pixels can then be fed back to the Java part for the DL model's evaluation.
- After evaluation, the results returned to the user by the application's UI is the DL model's top three predicted values of the resistor, and how much is the model certain about its prediction.

[1] Native here means written in a different programming language wrapped in *Java*

- The DL model itself is built and trained in Python using a Machine Learning framework or library.

4.2 Development Environment

Developing any software needs certain environments to edit, compile, and build the different code components. Software that provide these functionalities are called *"Integrated Development Environment"* or IDE.

The development process can take place in a more "raw" manner, i.e. without using an IDE, by using any code/text editor and install the dependencies and do what a modern IDE does manually. But of course this would make the development process take more time and effort to carry out.

An IDE provides many development-related tools, libraries, and frameworks in one *Graphical User Interface*. This enables the developer to focus more on the code itself, and thus, since the application around-which this work revolves has multiple software components[2], two IDEs were utilized:

1. *Android Studio* for developing the Android application's User Interface: The most widely used IDE when it comes to Android development. It was developed by *Google* as a fork of the infamous *IntelliJ* IDE from *JetBrains*. It provides a package manager that simplifies dependency installation and it supports hosting Android Emulators which are crucial for testing purposes [2].[3]

2. For this work, the desired development environment was a a local one. The *Windows Powershell* was used in administrator mode as a terminal to run various commands and as a running environment for training along with the *pip* Python package manager which was used to download dependencies like *Tensorflow*. *Microsoft Visual Studio Code* editor was utilized to write the python code that constructs and trains the model. Another IDE can be utilized, Anaconda for example. But developing and training the model of this work doesn't include a lot of debugging and the developer chose the lighter and simpler *Microsoft Visual Studio Code*. It is useful to also mention the relevant local development hardware and system as well:

[2]Such as the DL model and the Android application that is going to communicate with it
[3]The version used is: Android Studio Chipmunk | 2021.2.1

- Operating System: Microsoft Windows 10 v21H2 64-bit Operating system, x64-based processor.
- Processor: NVIDIA GeForce GTX 1060 with 6 GigaByte of dedicated RAM.[4]

It significantly cuts training time to use a GPU instead of a CPU. In fact, for this work, the time saved was 31.4%.

Google Colab can conveniently be used for developing the Deep Learning model: A *Google Research* product and it is a dedicated online IDE tailored for building and running *Python* code. It is also especially well-suited to Machine Learning applications[5].

There are of course many other alternative IDEs, such as *Mircosoft Visual Studio* or *Eclipse*. *Mircosoft Visual Studio* in particular has a big developer community behind it, in addition to direct support for several programming languages like *C/C++*, *Python*, and *Java*. The main reason why this work was implemented using *Android Studio* over *Mircosoft Visual Studio* is because for Android development, *Android Studio* has significantly higher support. But both can be really helpful and reliable for programmers. The author of this work also had more Android development familiarity with *Android Studio* therefore it was the choice.

4.3 Programming languages, libraries, and frameworks

Chapter 2 gave a theoretical overview of the terms and topics that are in some way or another a part of the system that this work aims to create. This section outlines the programming languages that were chosen to be a part of the implementation of the software which realizes the requirements laid out in Chapter 3. It will also discuss the reasons behind choosing one programming language over another, in addition to discussing some of the possible alternatives.

Figure 4.1 describes the design that was chosen by the developer to carry out the development of the Android application.

All these components will be later explained in this chapter.

[4]GPU specifications url:
https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1060/specifications/
[5]link to the webapp: https://colab.research.google.com/

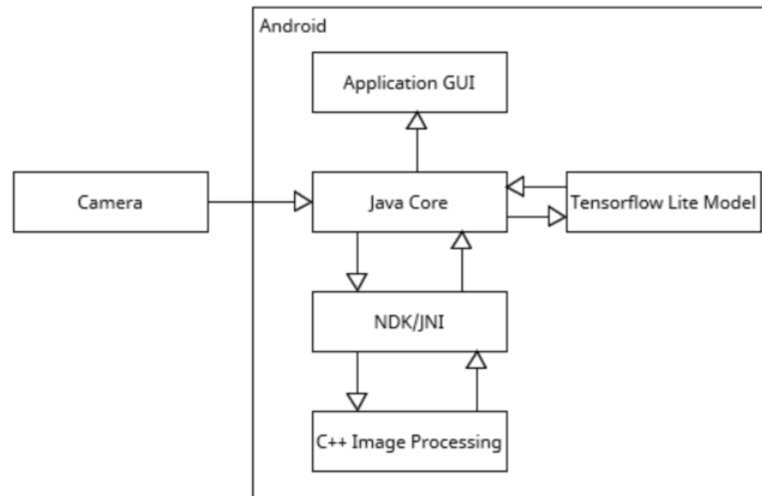


Figure 4.1: A visualized design of the application.

4.3.1 Python

When dealing with a certain complex or research-oriented applications such as the case with machine learning or deep learning, it becomes considerably helpful when using tools and higher-level languages and frameworks that facilitates the development process. But as with most software, the same systems can be developed with several languages, but opting for development using a low-level language, for example, can make the entire process unnecessarily more complex, difficult, and time consuming.

Python is a high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. [10]

For the purposes and requirements of this work, *Python* was the most compelling pick for building and training the DL model that is responsible for evaluating the input images.

Reasons for picking Python for the Deep Learning model development

1. *Python* has a simple syntax and a great readability, thus it is relatively easy to learn and use.
2. *Python's* boasts an extremely high-level structure and compilation environment, meaning that it allows a developer to focus more on the design flow and less on writing high-level function to compensate for the lack of them in other programming languages.
3. *Python* has a big vibrant community behind it, especially in research domains due to its syntactical simplicity. This helps in finding more technical support online or in academia when faced with a problem during development.
4. *Python* comes with a set of libraries[6] which make *Python* very suitable to handle and manipulate different data types like images. This, among the other reasons, explains why *Python* is widely used in research and in data science today.
5. Support for a large variety of DL libraries.
6. Python is a scripting language, meaning it can automate tasks execution in a runtime system. Therefore, scripting languages are usually interpreted at runtime rather than compiled.

The Python version used for this work is *Version: Python 3.9.13*. The relevant affiliated libraries and frameworks used will be mentioned because the compatibility between the versions used is relatively strict. For example, the local device's GPU was not detectable and therefore not usable by the interpreter while using the latest versions of the different libraries involved, specifically *Cudnn/Cuda*, *Tensorflow*, and *Python*. The versions used were:

1. *Python 3.9.13*
2. *Tensorflow 2.9.1*
3. *Tensorflow-gpu 2.6.0*
4. *Keras 2.9.0*

[6]Such as the *NumPy* mathematical library, and the *matplotlib* for graphs and plots

5. *Nvidia Cuda 11.7.0* [7]

6. *NumPy 1.22.4*

4.3.2 C++

C++ is a general-purpose programming language. It has imperative, object oriented and generic programming features, while also providing facilities for low-level memory manipulation. The *C++* programming language was designed and implemented by the Danish computer scientist *Bjarne Stroustrup* and it emphasizes performance and efficiency at execution time in addition to deeper control over devices hardware resources. [6, 24, 27]

The purpose behind introducing *C++* into this work's development process is due to the work author's personal preference and interest in promoting *C++* and contributing to its community, in addition to making use of its execution speed and its capable image processing and machine learning libraries such as *OpenCV* and *Tensorflow* respectively.

The *C++* part of this work is there to efficiently host and run the image processing functions. In this case, this means converting and reshaping the Java *Image* datatype into an *RGB Bitmap* in order to be able to feed the pixels to the DL model. This way, the *C++* section of the code plays the role of efficiently processing the *Java* frames. The *C++* part of this project is also encapsulated with the Java Native Development Kit or *NDK*, implementing the *JNI* or *Java Native Interface* in order to establish the connection between the two languages. This is visualized in Figure 4.1.

The Clang compiler was used for compiling the *C++* part of this work.

4.3.3 Java

Java is a high-level, class-based, object oriented programming language. It was the official preferred language for *Android* development until May 2019 when Kotlin[8] was introduced by *Google*. [2, 18]

[7] Cudnn, which exists within the Nvidia Cuda toolkit, is a GPU-accelerated library that allows for using GPU for the training process.
[8] Which is largely very similar to Java

Java was chosen for this work to be the development language of the Android application instead of *Kotlin* because the author of the work has some previous experience in working with *Java*. *Java* also provides the previously mentioned *JNI* which, in the context of the application at hand, allows for the outsourcing of the most processing-intensive part of the application which is the image processing part and enables the utilization of the *C/C++* high execution speed, making the entire application more efficient performance-wise.

For this application, the *Tensorflow Lite* mobile library was used for deploying or hosting tensorflow models on mobile devices.

The local Java Development Kit or JDK installed is *JDK 18.0.1.1* and the application was developed for a target *Android API* of level 25 or above.

4.3.4 Alternatives

For Machine Learning and Deep Learning applications, the options are becoming more numerous as time goes by. *Java* is another popular option, *Javascript*, and of course *C/C++*.

The choice of which programming language, or which combination of them, to be used in the development of software is in many cases adaptive and dynamic, and can in some applications just boils down to personal preference and experience.

4.3.5 Dataset

An essential part of any DL model training is the dataset used. For this particular application, no reliable dataset was found online therefore a dataset had to be created. The images should be taken from different angles, positions, and lighting conditions in order to reduce the chances of overfitting and increase the capacity for generalization.

Images of the dataset can have different formats, colored or grayscale depending on the application, but for the purposes of this thesis RGB formats are necessary because the classification takes place on the bases of the colored stripes on a given resistor.

In the next Chapter 5, the creation of the dataset will be further discussed.

4.3.6 Approach

Several approaches can be applied to tackle the image classification problem at hand. Using a pre-trained model within a *transfer learning* paradigm that can identify and crop the relevant part of an image in order for our model to be trained on classifying the output of such models. The author opted not to use such pre-trained models and instead train a CNN model to handle the images as inputs directly and make predictions based on it. This decision was made because the developer wanted to explore how well a plain convolutional network can perform in the given task without external interference or simplification.

Figure 4.2 outlines some of the possible approaches mentioned earlier in this section. Keep in mind that CNNs can have dense layers as final layers that are responsible for classifying a flattened input coming from convolutional layers. In such cases, the convolutional layers are responsible for feature extraction while the final dense layers are responsible for classifying the convolutional layers flattened output.

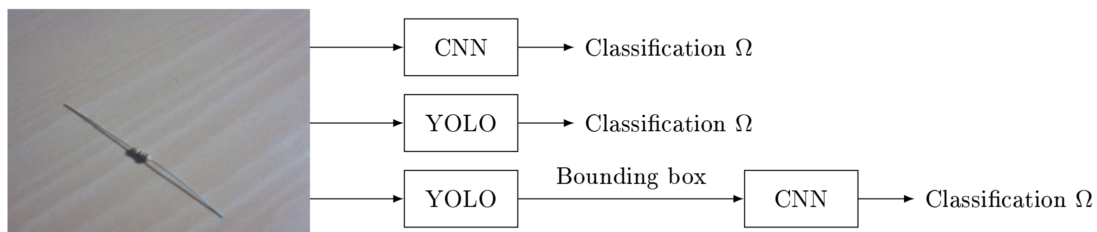


Figure 4.2: Some possible approaches for implementing the application.

5 Implementation

Achieving the requirements of this thesis can be done in many different ways using various tools, designs, and approaches. But in the circumstances of a thesis, such as the time and hardware and other resource limitations, several trade-offs had to be made and a specific approach was devised.

5.1 Training

This section demonstrates the technical and practical steps that are relevant to the creation of the dataset and developing the model that will train using the produced dataset.

5.1.1 Creation of the dataset

For training the network model, a labeled resistor images dataset had to be used, and because no dependable one could be found online a new dataset had to be produced. The *createDataset.py* Python script was used to serve this purpose as it allows for the capturing of images and saving them to the hard disk.

The script creates folders named with the corresponding resistor value because it is a convention that the folder names for the different classification classes correspond to the labels and thus the resistor values.

The script uses the *OpenCV* Python library's default capture resolution for the images and that resolution is $(width, height) = (640, 480)$. This resolution is relatively large for a typical deep learning model input and thus this particular issue later proved to be a significant factor in further limiting the development and training options and characteristics.

Figure 5.1: Sample images of 150 Ω resistor from the produced dataset.

The camera used to capture those images was a typical laptop's embedded webcam and this is suspected to be a major factor in the low quality of the captured images of the dataset, along with less significant factors like the developer's need for a relatively large number of images in a short amount of time.

A good and effective training process depends on a good quality and a well engineered dataset, therefore dataset preparation is by itself a large topic to dive into. With this application as an example, engineering some *"noise"* in the captured images can result in a more "healthy" training process and can lead to better results as it allows for better capacity for generalization among other things. But having too much noise in a dataset is a recipe for a failed training.

Figure 5.1 demonstrates a selection of sample images from the produced dataset.

As Figure 5.1 outlines, images vary in lighting conditions, distance from camera, background, and rotation. With about 32000 thousand images in the produced dataset, this can have a good impact on the model's ability to generalize.

5.1.2 Model development

For this work's DL model's training all the factors mentioned above were very clearly manifested in the dataset and training performance for several different models[1]. The total number of images produced was close to 44,000 images, which was later reduced to about 32,000 after removing particularly noisy and vague images. Within every classification class, i.e. for every single folder in the dataset, the images were taken with different angles, backgrounds, zoom, and lighting conditions in order to reduce the chances of overfitting and increase the capacity for generalization. The dataset's training/validation split was 75%/25% respectively in accordance with deep learning conventions and recommendations laid out in [15]. The end result as far as this dataset goes, was enough to get some relatively acceptable results but the author of this work has doubts about the impact of the shortcomings of this dataset on the "real" and practical performance of the model within the Android app.

A training script was devised as well called *model-trainer.py* which contains:

1. The Python code that constructs the network model using *Keras* and compiles it.
2. Dataset handling and other dataset operations like data augmentation as shown in Figure 5.2.
3. The training code along with training configurations and callbacks. Figure 5.3 outlines this part of the script.
4. The code which is responsible for saving the model and converting it to a *.tflite* file in order for it to be imported and used in the *Android* application. This part is shown in Figure 5.4.

Hyper-parameters were initially chosen in accordance with the general guidelines of DL model development online and in [15].

- The learning rate was chosen to be $\eta = 0.001$ which is the default learning rate for the *Adam* optimizer that was used in all trained models. The value of the learning rate was reduced by an order of magnitude when *model 1* performance was not improving as desired. This reduction to $\eta = 0.0001$ made a 17% improvement in validation accuracy for *model 2* as can be seen in Figure 5.5.

[1]Several models were trained for this work in the process of finding the best fit

```python
# Setup paths to our data directories
train_dir = "resistors/train"
test_dir = "resistors/test"

# Import data from directories and turn it into batches
train_data = train_datagen.flow_from_directory(directory=train_dir,
                                               batch_size=16,
                                               target_size=(640, 480),
                                               class_mode="categorical",
                                               seed=77)
valid_data = valid_datagen.flow_from_directory(directory=test_dir,
                                               batch_size=16,
                                               target_size=(640, 480),
                                               class_mode="categorical",
                                               seed=77)
#print(train_data[0])

# Create an augmented data generator instance
test_datagen_augmented = ImageDataGenerator(rescale=1/255.,
                                            rotation_range=0.15,
                                            width_shift_range=0.15,
                                            height_shift_range=0.15,
                                            zoom_range=0.15,
                                            horizontal_flip=True)

test_data_augmented = test_datagen_augmented.flow_from_directory(test_dir,
                                                                 target_size=(640, 480),
                                                                 batch_size=16,
                                                                 class_mode="categorical")
```

Figure 5.2: Code snippet for data handling and preparation in model-trainer.py script.

- Introducing batch normalization and optimizing its momentum caused a further 11% increase in validation accuracy as shown in Figure 5.6.

- Data augmentation, i.e. random rotation, reconstrast, flipping, and zoom, further improved the validation accuracy by about 16%. This was coupled with optimizing normalization momentum and batch size from 16 to 24 in order for a more effective batch normalization process. The improvement can be seen in Figure 5.7.

- *ReduceLROnPlatue* callback was introduced in *model 9* and it increased the model's validation accuracy by about 4%. Figure 5.8 demonstrates this for *model 9*.

- The resulting validation accuracy after *model 9* started stagnating around the 90% mark. Changes to the model's structure, resolution, and hyper-parameters like the learning rate, batch size, and number of epochs produced no noticeable improvement. This can be seen in Figure 5.9.

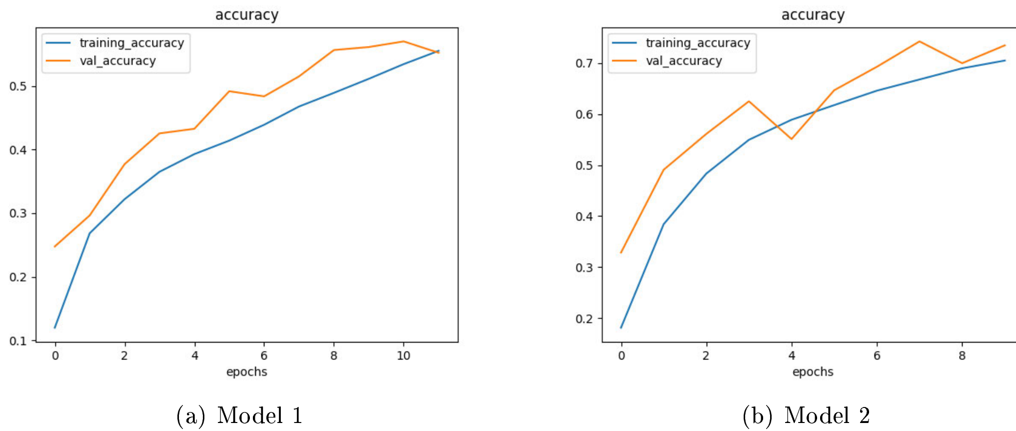
```
186  # Fit model
187  history_1 = model_1.fit(
188      train_data,
189      epochs=30,
190      batch_size=16,
191      callbacks=[   # keep in mind that some callbacks are somewhat RAM and CPU/GPU expensive
192          tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=6),
193          tf.keras.callbacks.ModelCheckpoint("trained_model_checkpoint"),
194          tf.keras.callbacks.ReduceLROnPlateau(
195              monitor='val_loss',
196              factor=0.1,
197              patience=3,
198              mode='auto',
199              min_delta=0.0001,
200              min_lr=0.0000001,
201          )
202      ],
203      steps_per_epoch=len(train_data),
204      shuffle=True,
205      validation_data=test_data_augmented,
206      #validation_data=valid_data,
207      validation_steps=len(test_data_augmented))
```

Figure 5.3: Code snippet for model fitting and callbacks in model-trainer.py script.

```
210  # Save  model
211  model_1.save("trained_model_1")
212  model_1.save("trained_model_1.h5")
213
214  # convert model to tflite
215  converter = tf.lite.TFLiteConverter.from_saved_model("trained_model_1")
216  tflite_model = converter.convert()
217
218  with open('model.tflite', 'wb') as f:
219      f.write(tflite_model)
220      f.close()
221
222  # convert model to tflite
223  converter = tf.lite.TFLiteConverter.from_saved_model("trained_model_checkpoint")
224  tflite_model = converter.convert()
225
226  with open('modelCheckpoint.tflite', 'wb') as f:
227      f.write(tflite_model)
228      f.close()
```

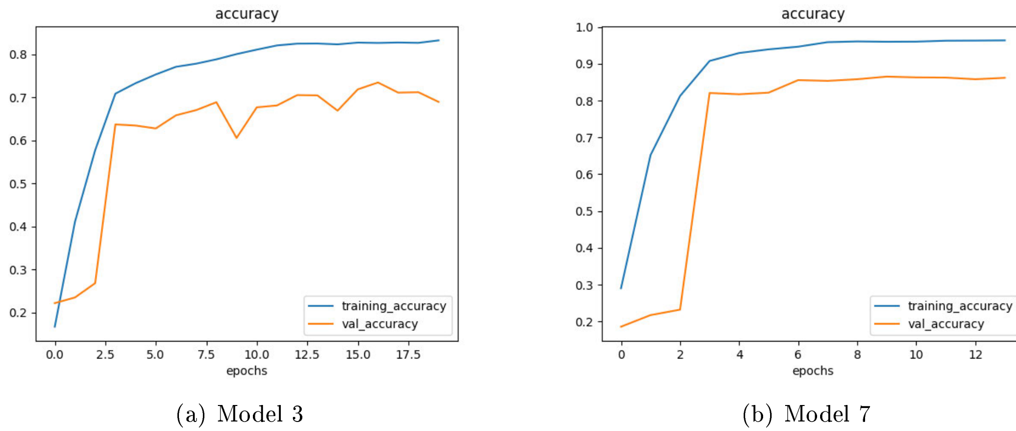
Figure 5.4: Code snippet for the model saving process in model-trainer.py script.

The training process is initiated by a Windows Powershell with administrator permissions. The *cd* or *change directory* command can be used to access the project folder where the *resistors* folder that contains the dataset is located along with the *model-trainer.py* training script. Then, the training process is launched by typing the command *Python model-trainer.py*.



(a) Model 1 (b) Model 2

Figure 5.5: Model 1 and 2 accuracy plot comparison

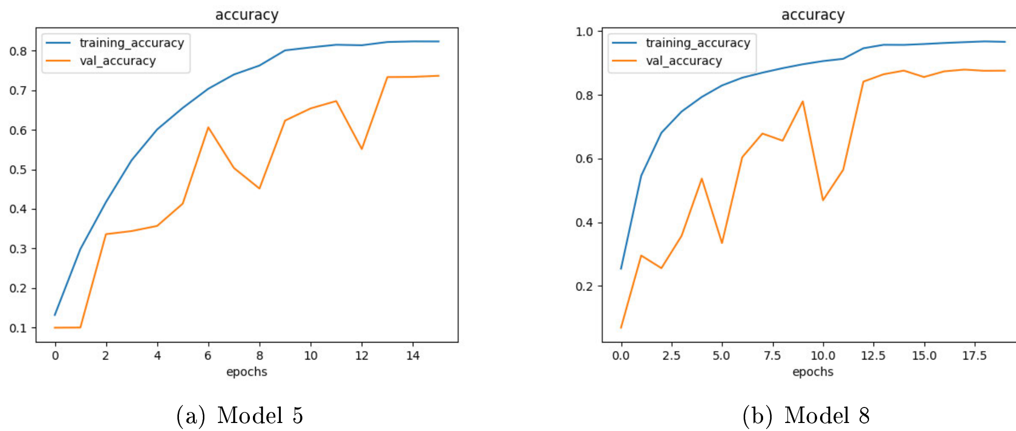


(a) Model 3 (b) Model 7

Figure 5.6: Model 3 and 7 accuracy plot comparison

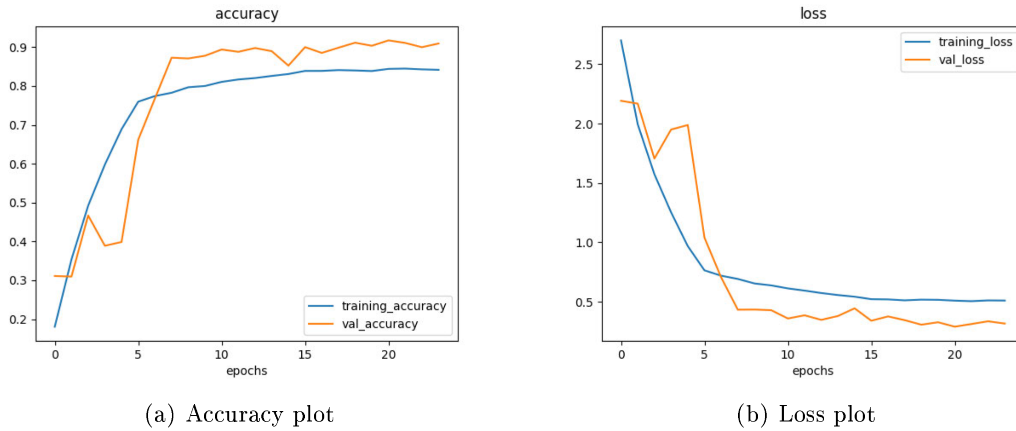
The amount of time that training the model takes can vary according to several factors like hardware, number of parameters, and the size and type of the dataset. On average, and in the context of this work, a single epoch of training was about 20 minutes.

When the training process is concluded, the script automatically saves the model and further information, e.g regarding weights and other parameters, in the project path or directory. The saving takes place in several formats for the model: The default one, the *.h5* format, and the *.tflite* format which can be imported and used in Android applications.



(a) Model 5 (b) Model 8

Figure 5.7: Model 5 and 8 accuracy plot comparison



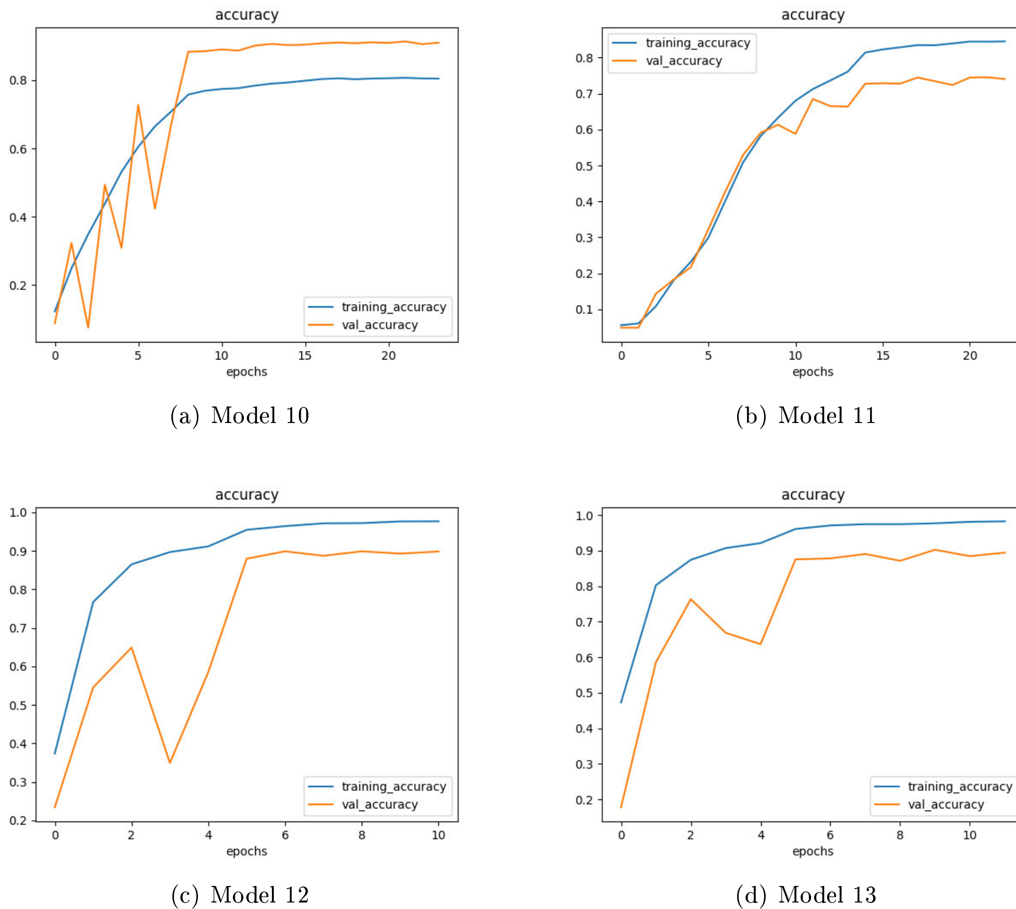
(a) Accuracy plot (b) Loss plot

Figure 5.8: Model 9 accuracy and loss plots

Appendix A.2 draws some comparison between the trained models and provides each model's summary, accuracy plot, and loss plot. The developer followed the recommendations laid out in Chapter 2. Models were trained sequentially starting with *model 1* as the first trained model, *model 2* was second, and so on.

Observations that the developer had during the training of the models are:

- Batch normalization was introduced starting with model 7 to every conv-pool block before the pooling layer. It had a positive impact on the validation accuracy of the trained models. It is difficult to precisely measure the improvement but contrasting



(a) Model 10

(b) Model 11

(c) Model 12

(d) Model 13

Figure 5.9: Model 10, 11, 12, and 13 accuracy plot comparison

model 6 and *model 7*, the improvement was about 10%. An important note is that batch normalization is not effective when using small batch sizes such as a batch size of 8 because it relies on calculating the mean and standard deviation of every batch, thus the larger the batch size the better normalization metrics are. [23]

- Randomly augmented data was introduced to the dataset starting from *model 9*. The developer couldn't observe any noticeable impact on the performance of the models that it was applied to. But like creating the dataset, data augmentation should be carefully planned and implemented.

- Dropout was also introduced in *model 13* and it had no significant impact on the performance of the model in comparison to remaining models.

- The developer observed a relationship between overfitting, total number of training epochs, and the learning rate. This relationship is not concrete and it could be a result of correlation by the developer. The relationship is that when a model is trained with a high number of training epochs while having a large learning rate like 0.01, chances of bad performance and overfitting occurring are increased especially when the model is too complex or has a large number of trainable parameters.
- The optimal range for the number of trainable parameters in a model that was observed by the developer for the application of this thesis is between $300,000$ and $1,500,000$.

Utilizing batch normalization and dropout can sometimes necessitate increasing the total number of epochs or the learning rate or both. This is because those two techniques play a role in avoiding overfitting and increasing a model's ability to generalize over unseen data by trimming the amount of learning done by the model so it doesn't obtain a strong "opinion" in a single training epoch.

5.2 Integration into the Android application

Embedding the trained model into the Android application involves using the *.tflite* model file format and including it in the project's path. Android Studio makes this process extremely straightforward. After the inclusion of the model file, it is treated as a Java class object and allows the developer to pass processed RGB Bitmap images to the model and then access the results or output of the evaluation carried out by the model as shown in Figure 5.10.

From the requirements laid out in Chapter 3, there were many different design options to implement the functionality that achieves those requirements. The chosen design was as follows: Figure 5.11 is the class diagram for this application. It outlines the methods and member functions relevant to the core functionality of the application.

Some visualization helps with understanding the structure and functionality of the system so a class diagram for the system will be provided in Figure 5.11 and an activity diagram will be provided in Figure 5.12.

```
139    // Passing the prepared ByteBuffer to the model as input
140    inputFeature0.loadBuffer(byteBuffer);
141    // Runs model inference and gets result.
142    Model.Outputs outputs = model.process(inputFeature0);
143    TensorBuffer outputFeature0 = outputs.getOutputFeature0AsTensorBuffer();
144    // Get model labels
145    List<String> labels = FileUtil.loadLabels(getApplicationContext(),  filePath: "labels.txt");
146    /*labels.forEach(label -> {
147        Log.e("label: ", label);
148    });*/
149    // labels the first axis with size greater than one
150    TensorLabel labeled = new TensorLabel(labels, outputFeature0);
151    // If each sub-tensor has effectively size 1, we can directly get a float value
152    // certainty map with the model's labeled choices and how confident it is
153    Map<String, Float> certainty = labeled.getMapWithFloatValue();
```

Figure 5.10: Code snippet for model access in CameraAccessActivity.java class.

- MainActivity.java class: contains the main root of the application, in addition to the necessary functions that would be called in its corresponding *activity_main.xml* which is the file responsible for the visual markup design and layout of the corresponding java class file i.e. this file represents the main UI. The UI contains a greeting activity with some instructional text and a button called "Open Camera" which, when clicked, configures and initializes the camera by creating an intent for it, and requests the necessary permissions for using the device's camera, then it takes the user to the activity responsible for the resistor classification process, and that activity is *CameraAccessActivity.java* and its corresponding markup file *activity_camera_access.xml*.
- CameraAccessActivity.java class: contains the functions that are relevant to the classification process and operating the camera. Its corresponding markup file *activity_camera_access.xml* contains the top three results text element along with a camera preview element that will display the camera view feed on the UI. The critical methods that are concerned with the classification process are:
  1. bindImageAnalysis: sets up and initializes the camera and creates a camera view and binds it with a "camera view preview" which exists in the markup file *activity_camera_access.xml*. It also creates and binds an image analysis "use case" to the functionality in order to allow for operations on the frames coming from the camera feed. This function also calls the functions "toBitmap" which

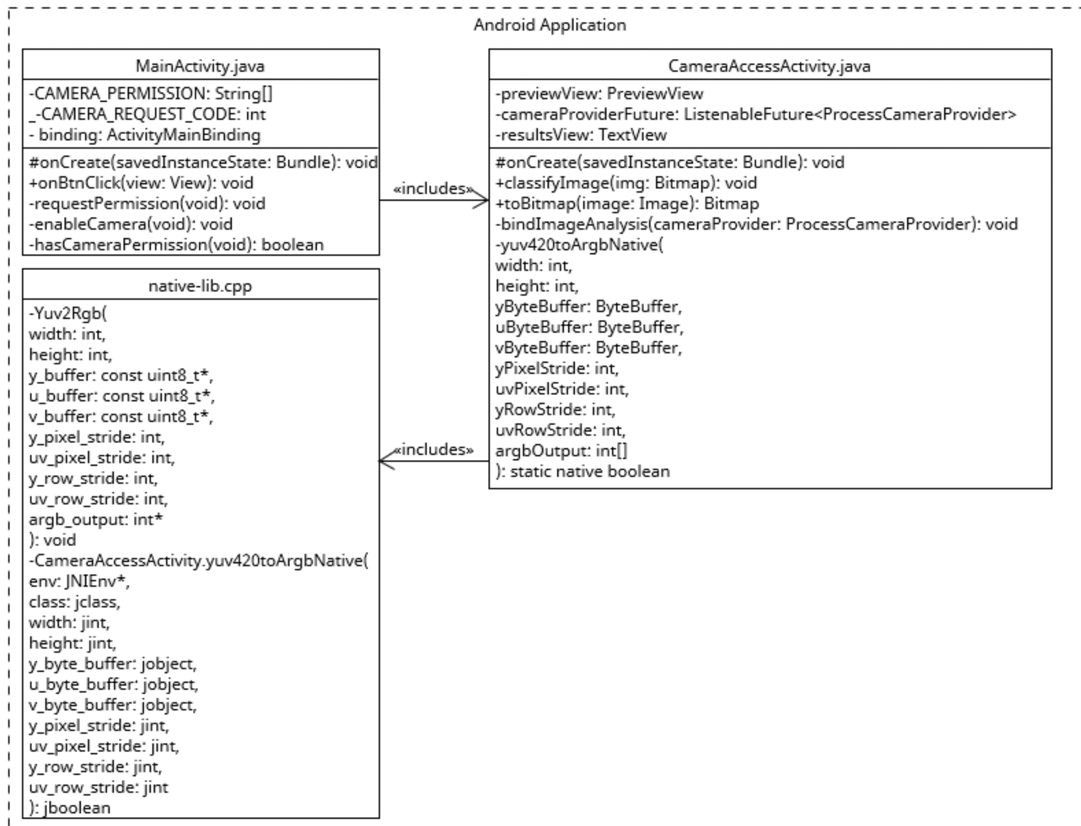


Figure 5.11: Class diagram of the Android application with relevant methods.

is responsible for image conversion into RGB bitmaps, and the “classifyImage” function which is responsible for classifying the processed images.

2. toBitmap: takes an “Image” type and returns a converted RGB bitmap after calling a native *C++* image processing function in the *native-lib.cpp* native library file. It is worth mentioning that the native library here carries out the calculations in place in RAM.

3. classifyImage: this method instantiates a tflite object and creates the necessary “bytebuffers” to allow communication with the deep learning model so it can be utilized to classify images. After feeding images to the model and receiving the results back, the method then will display the results on the UI through the bound text view after qualifying the confidence level of the model in its choices.

- native-lib.cpp: The application's native *C++* part is represented in this library. It implements the *Java Native Interface* or *JNI* and it is linked with the *CameraAccessActivity.java* class. The library contains the functions that are relevant for converting *Image* data type frame coming from the camera feed to an RGB *Bitmap*. All calculations relevant to a frame's pixel data are done in memory directly, i.e. in place.

Figure 5.12 visually describes the steps in the application's functionality in an activity diagram. It abstractly outlines the steps that the application takes when the user opens the camera feed and starts the classification process.

It is worth noting that a single classification loop, shown in Figure 5.12, achieves the requirement from Chapter 3 which states that the application should classify an image within two seconds. In the case of the testing Android device[2] this classification process takes milliseconds for a classification cycle to be concluded.

It should be mentioned that as with the rest of modern software development, Android development involves stating required permissions in the Android manifest markup file and the dependencies and build configurations for the *Gradle* build tool that is used to build the application. Of course, *Android Studio* makes it so that build-related processes such as this process or the native code building and linking become significantly more straightforward.

Figure 5.13 demonstrates snapshots directly from the Android application GUI. It helps to visualize the final shape of the application on an Android device.

[2]Huawei nova 5T

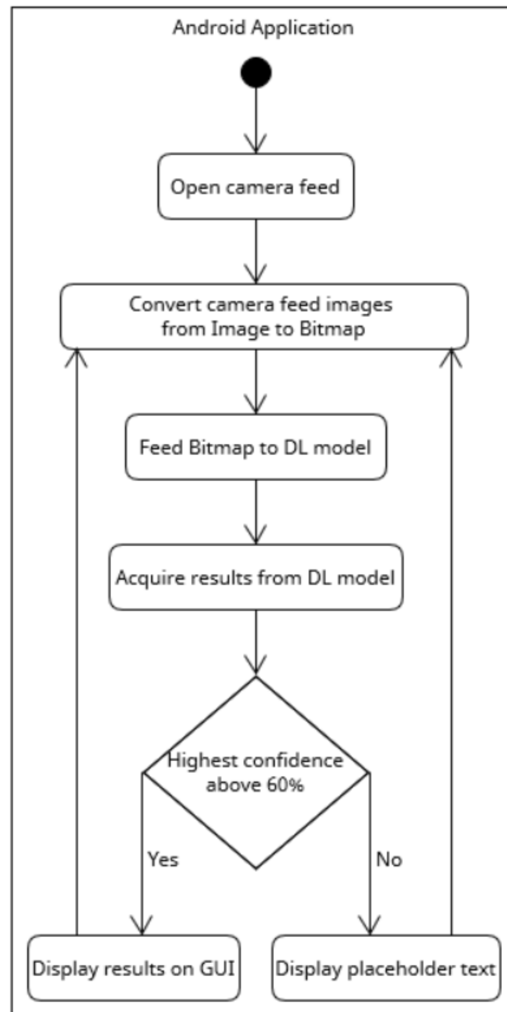
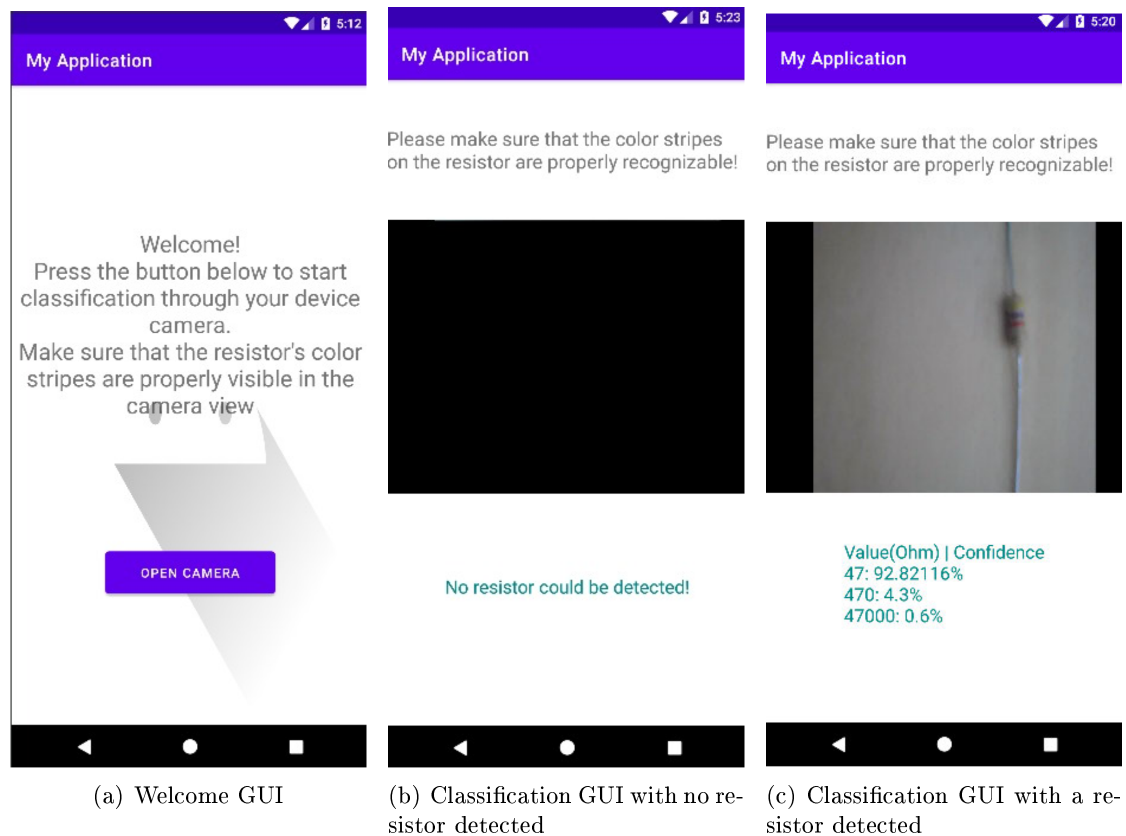


Figure 5.12: Activity diagram of the Android application.



(a) Welcome GUI

(b) Classification GUI with no resistor detected

(c) Classification GUI with a resistor detected

Figure 5.13: Snapshots of the application running on an Android device.

6 Project results, insights, and conclusions

In this chapter, some of the developer's/author's evaluations, conclusions, and learned experiences will be discussed. Where does the developer believe that there can be room for improvement, what alternatives can be utilized to enhance and expand the application, and what are some of the shortcomings, mistakes, limitations, problems, and obstacles faced during the development process and where do they originate. In addition, to what extent the developer thinks those problems are solvable or avoidable or reducible and if so then how.

As mentioned previously on several occasions in this documentation, the main problem for development was the developer's lack of experience in both pillar topics of this work, which are Android development and deep learning. There is also time management issues and student status elated issues as well, but the discussion of problems in this chapter will not reiterate this and will focus on the more technical aspects of the development conclusions.

6.1 The training process

Developing and training the model are the main pillars of this work, and this documentation correspondingly focuses on that process. This section will explore the developer's main conclusions regarding the training process.

Throughout the training part of this work, several models were trained. The models vary in accuracy, number of parameters, resolution, learning rate, structure, callbacks, and techniques used such as batch normalization, data augmentation, and dropout. Each trained model's summary along with its accuracy and loss plots are briefly demonstrated and discussed in appendix Section A.2.

6.1.1 Problems

The main issues regarding training the deep learning model in Python were:

1. The large number of options and choices to implement, build, and train a deep learning model was at first a disadvantage because it makes it difficult to be relatively certain about the efficacy of the chosen development choices.

2. The complex nature of a successful training process for a deep learning model.

3. Computer vision applications for deep learning means that the training process will become significantly more resource-expensive and will take a lot more time to be concluded. This also introduces obstacles and issues that are not necessarily there in other types of deep learning applications because images are particularly resource-expensive when it comes to data types. This limits the available options for development even further, this work being in the context of a Bachelor thesis with limited and relatively short amount of time.

4. Progress is relatively slow because the effects and results of training can only be properly observed and evaluated after the training is over, making model enhancements take large amounts of time to manifest.

5. The training was done with *Keras* in Python and compared to other frameworks it is relatively well-documented and has a larger online community than others, but it can help if this community was even larger and more active.

6. The limited resources, hardware and other types, that were available during training was a noticeable issue.

7. The relatively strict nature of frameworks compatibility was a major obstacle in the beginning and it made preparing the development environment unnecessarily more difficult.

8. The still unexplained variation between the model's validation accuracy during training and the real practical accuracy when integrated into the Android application. Although the author speculates that this divergence between the validation accuracy and the real practical accuracy of the model is mainly due to the bad quality of the dataset used, i.e. the camera used for the creation of the dataset differed significantly from the camera of the testing Android device. It could be

worth noting that the testing device's camera produces images with significantly better quality than that of the camera that was used to create the dataset.

6.1.2 Insights

This section will layout some of the author's insights regarding the training process, although currently many of these insights can only be based on rational speculation.

- Regarding the discrepancy between validation accuracy during training and real practical accuracy in the application, it is believed that the main reason behind it is the low quality of the dataset, whether it is too much noise or the images themselves were of bad quality or unrecognizable.
- As mentioned previously in this chapter, the image-related nature of the application imposes limitations, some of which are solvable by simply using a more powerful, higher-performance device to train the model. This alone can cut training time significantly and thus allows for more time for development.
- Using a better camera, technique, and better lighting conditions in the dataset creation process could've played a major role in improving practical accuracy and performance altogether.
- Training a model using *C++* wouldn't have an impact on training time. This is because *Tensorflow* and the *Python* code call *C++* libraries thus taking advantage of its high execution speed.
- The models built and trained by the developer varied in architecture and structure, and in the number of parameters involved. The developer in this regard was limited to a maximum of several million parameters otherwise the system would run out of GPU memory and crash.
- Having the resolution of input images during training equaling half or even less[1] of the currently used 640×480 resolution can greatly affect the training efficiency and allows for more room for the developer to maneuver and further manipulate the model's architecture and number of parameters. This can be done by processing the images, using for example an image processing library like *OpenCV* or a previously trained transfer learning model, and down-scaling them or simply cropping the

[1] A typical image size for deep learning models is 224×224.

relevant parts of the image, resulting in images that contains a higher portion of relevant information that the model can learn and use to predict. It should be noted that changing the resolution or size of the model's input must be accompanied in carrying out the same transformation in the Android application as well.

- Instead of using RGB image format, perhaps using a more condensed and less scattered representation or format like the *Hue*[2] color channel, can offer the model the information in a more consolidated manner and that can have a positive impact on the efficacy of the training process, reducing training time and number of model parameters.

6.2 The Android application

The Android development part of this work was not as vague and foggy as it was the case for training the deep learning model. It relied more on traditional software development and thus the developer could manage the development process in a more structured manner.

6.2.1 Problems

The Android development process was not without obstacles and issues, some of which are the following:

1. Google dubbing Java as not being the official supported language of Android development anymore and replacing it with *Kotlin* had a big impact on modern Android development using Java. Less documentation and support content, and a smaller online community made development unnecessarily complicated.

2. The performance[3] of the application was acceptable but it is clear that there is still room for improvement.

[2]Hue format can typically be represented by a single number, often corresponding to an angular position around a neutral point or axis on a color space coordinate diagram, or by its dominant wavelength or by that of its complementary color. [3]

[3]The time-related performance and not the accuracy.

3. The rate of time consumption that was dedicated to the training process made the time that was appropriated for the application development relatively smaller than what the developer planned.

6.2.2 Insights

This section will now go over some suggestions by the developer for the improvement of the application in general terms:

- Moving more of the Java parts and functionalities of the application to the native library can further improve the time performance of the application.
- Using a reliable image processing library for image processing instead of the manual code that was written.
- Using a different image format like *Hue* can also have an impact here in the application, possibly improving both time and accuracy performance.
- Using *Kotlin* can significantly expand development choices for Android, especially for inexperienced developers.

6.3 Conclusions

Some of the conclusions were already stated in this chapter, but this section will briefly discuss what other possible factors or conditions that can affect the development of similar kinds of applications. This section will also be concluded with the author's verdict on the application and the prospects of its further enhancement and expansion, and some deductions that can be learned and used to build better applications of this kind in general.

6.3.1 The deep learning model

The points regarding the training of deep learning models that the author of this work would like to emphasize according to the author's learned experience during the course of this thesis are:

- Training deep learning networks within applications that have to do with images is time consuming and resource expensive. Care should be given to this fact when trying to manage the limited development time in a more efficient manner.
- The quality and quantity of the dataset are the main pillars of a successful training process. Special care should be given to making sure that the dataset provides enough information with a "healthy" amount of randomness and noise for generalization purposes. If a dataset for the desired application couldn't be found in online resources then creating a new reliable dataset can be a tedious endeavour.
- The training process involves a fair amount of trial and error. The trial and error process should be guided by a good understanding of the application and how it should learn and function, and a good visualization of the model's structure and inputs.
- Using callback functions or not can depend on the type of application at hand, but three important callbacks that can always be useful in all applications are the "ReduceLROnPlateau", the "EarlyStopping", and the "ModelCheckpoint" callbacks. Those callbacks have proved to maximize time-related efficiency.
- Following some of the guidelines, conventions, and insights of deep learning researchers can, particularly in the early stages of training, provide a good compass and put the training process closer to the right track.
- Employing batch normalization will require care with regards to the batch size, as batch normalization is less effective with smaller batch sizes. Recommended batch size is 32, but during the training process of this work, the developer on some occasions used a smaller size 16 in order to have more memory to allow for a model with a higher number of parameters.

6.3.2 The Android application

With relation to the Android development part of this project, the main point that the author found to possibly simplify the development of the application is to use *Kotlin*, especially for inexperienced developers. Another valid point is also to make sure to better understand how the *Android* operating system works and how the build process is carried out, as the author believes that the better this understanding, the easier the development process will be.

6.3.3 Working with images

The author found that dealing with image-related applications necessitates some foundational knowledge with images and image processing. This point was particularly clear during the development of the Android part of this work.

6.3.4 Final comments

The author believes that this particular application of classifying resistor values from images is possible and having a high practical accuracy is achievable. It may come down to using an image processing library to identify and crop the relevant parts of the resistor and passing it to the model, or simply having a more reliable and higher quality dataset.

The domain of deep learning and the scope and potential for its utilization is really fascinating and currently expanding. There already exist models that outperform human being in highly complex and essential applications in medicine, science, industry, and logistics. There is also the potential and prospects of *"Artificial General Intelligence"* or *AGI* which are the kind of models that were able to effectively and efficiently learn several complex tasks and human-like skills. However, those types of models are still theoretical.

Bibliography

[1] Aggarwal, Charu C.: *Neural Networks and Deep Learning. A Textbook*. Springer, 2018. – ISBN 9783319944630; 3319944630

[2] Allen, Grant: *Android for Absolute Beginners: Getting Started with Mobile Apps Development Using the Android Java SDK*. 1st ed. Apress, 2021. – ISBN 9781484266458; 1484266455; 9781484266465; 1484266463

[3] Burger, Wilhelm ; Burge, Mark J.: *Digital Image Processing*. 2nd edition. Springer, 2016

[4] Ceron, Rodrigo: *AI, machine learning and deep learning: What's the difference?* https://www.ibm.com/blogs/systems/ai-machine-learning-and-deep-learning-whats-the-difference/. – [Online; accessed 27/May/2022]

[5] Chow, Tommy W. S.: *Neural Networks and Computing: Learning Algorithms and Applications (Series in Electrical and Computer Engineering)*. World Scientific Publishing Company, 2007 (Series in Electrical and Computer Engineering 7). – ISBN 1860947581; 9781860947582

[6] C++.org: *Welcome to C++*. http://www.cplusplus.org/. – [Online; accessed 23/May/2022]

[7] DeepAI.org: *What is a Perceptron?* https://deepai.org/machine-learning-glossary-and-terms/perceptron. – [Online; accessed 27/June/2022]

[8] Dictionary, Oxford Advanced L.: *Computer*. https://www.oxfordlearnersdictionaries.com/definition/english/computer. – [Online; accessed 27/May/2022]

[9] Economist, The: *Python is becoming the world's most popular coding language*. https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language. – [Online; accessed 24/May/2022]

[10] Foundation, Python S.: *What is Python? Executive Summary*. https://www.python.org/doc/essays/blurb/. – [Online; accessed 23/May/2022]

[11] Girshick, Ross ; Donahue, Jeff ; Darrell, Trevor ; Malik, Jitendra: *Rich feature hierarchies for accurate object detection and semantic segmentation*. https://arxiv.org/abs/1311.2524. – [Online; accessed 24/May/2022]

[12] Google: *Google's Python Class*. https://developers.google.com/edu/python. – [Online; accessed 24/May/2022]

[13] Hensel, Marc: *Deep Learning Practitioner (working title)*. HAW Hamburg, to be published. – http://www.haw-hamburg.de/marc-hensel

[14] Kingma, Diederik P. ; Ba, Jimmy: *Adam: A Method for Stochastic Optimization*. https://arxiv.org/abs/1412.6980. – [Online; accessed 27/June/2022]

[15] Krohn, Jon ; Beyleveld, Grant ; Bassens, Aglae: *Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence*. Pearson Education, 2019 (Addison-Wesley Data & Analytics Series). – ISBN 0135121728; 9780135121726

[16] Lee, Roger ; Kim, Jong B.: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Springer, 2022 (Studies in Computational Intelligence, 951). – ISBN 3030670104; 9783030670108

[17] Lewis, Tanya: *A Brief History of Artificial Intelligence*. https://www.livescience.com/49007-history-of-artificial-intelligence.html. – [Online; accessed 27/May/2022]

[18] Ogihara, Mitsunori: *Fundamentals of Java Programming*. 1st ed. Springer, 2018. – ISBN 9783319894904; 3319894900; 9783319894911; 3319894919

[19] Redmon, Joseph ; Divvala, Santosh ; Girshick, Ross ; Farhadi, Ali: *You Only Look Once: Unified, Real-Time Object Detection*. https://arxiv.org/abs/1506.02640. – [Online; accessed 24/May/2022]

[20] Ren, Shaoqing ; He, Kaiming ; Girshick, Ross ; Sun, Jian: *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. https://arxiv.org/abs/1506.01497. – [Online; accessed 24/May/2022]

[21] Rosenblatt, Frank: THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN. In: *Psycological Review* 65 (1958), Nr. 6, S. 386–408

[22] Silberschatz, Abraham: *Operating System Concepts*. Abridged 10th Edition. Wiley, 2018 (10). – ISBN 9781119456339; 1119456339

[23] Spencer, Quinn: *Neural Networks: Deep Learning and Machine Learning Outlined*. self-publ., 2018. – ISBN 1983255955; 9781983255953

[24] Stroustrup, Bjarne: *The C++ programming language*. 3rd ed. Addison-Wesley, 1997. – ISBN 9780201889543; 0201889544

[25] Tomsho, Greg: *Guide to Operating Systems*. 5th. Cengage Learning, 2016. – ISBN 1305107640; 9781305107649

[26] Tsui, Frank ; Karam, Orlando ; Bernal, Barbara: *Essentials of Software Engineering, 5th Edition*. 5. Jones & Bartlett Learning, 2022. – ISBN 9781284229004; 1284229009; 9781284228991; 1284228991

[27] Wikipedia: *C++*. https://en.wikipedia.org/wiki/C%2B%2B. – [Online; accessed 23/May/2022]

[28] wikipedia.org: *Integrated development environment*. https://en.wikipedia.org/wiki/Integrated_development_environment. – [Online; accessed 24/May/2022]

[29] wikipedia.org: *Linux*. https://en.wikipedia.org/wiki/Linux. – [Online; accessed 25/May/2022]

[30] wikipedia.org: *Machine vision*. https://en.wikipedia.org/wiki/Machine_vision. – [Online; accessed 27/May/2022]

[31] wikipedia.org: *Perceptron*. https://en.wikipedia.org/wiki/Perceptron. – [Online; accessed 24/May/2022]

[32] wikipedia.org: *Programming language*. https://en.wikipedia.org/wiki/Programming_language. – [Online; accessed 24/May/2022]

[33] WIKIPEDIA.ORG: *Supervised Learning*. https://en.wikipedia.org/wiki/Supervised_learning. – [Online; accessed 27/May/2022]

[34] ZHANG, Aston ; LIPTON, Zachary C. ; LI, Mu ; ; SMOLA, Alexander J.: *Dive into Deep Learning*. 0.16.1. d2l.ai, 2021

A Appendix

A.1 Back-propagation

Keeping Equation 2.1 in mind in addition to the fact that σ refers to the activation function, if L was to symbolize a model's final layer, and therefore $(L-1)$ would refer to the layer before last, and a^L is layer L activation and z^L is layer L output, i.e. each activation and output for each corresponding neuron in the given layer L. Then, for the final layer, Equation A.1 can be expressed. [15]

$$\begin{aligned} z^L &= w^L \cdot a^{L-1} + b^L \\ a^L &= \sigma(z^L) \\ C_0 &= (a^L - y)^2 \end{aligned} \tag{A.1}$$

C_0 is the final layer's cost and the first cost to be calculated, i.e. the root cost. After calculating it, with each layer back that is passed through, the gradient of the total error from the preceding layer $\frac{\partial C}{\partial a^L}$ is calculated, and in this way, the total error of the system is propagated backwards. [15]

This backwards propagating value of the final layer is referred to as δ_L and with every layer back this calculation is remade to produce $\delta_{(L-1)}$, and so on. So for the final layer the Equation A.2 can be expressed.

$$\delta_L = \frac{\partial C}{\partial a^L} = 2(a^L - y) \tag{A.2}$$

Now, in order to update the weights in layer L, it is necessary to find the gradient of the cost with regards to the weights $\frac{\partial C}{\partial w^L}$. And keep in mind that according to the chain

rule[1], this is equal to the product of the gradient of the cost for the preceding layer with regards to the preceding layer's output and the gradient of the activation with regards to z and the gradient of z with regards to the weights w of that same layer. All this is expressed in Equation A.3. [15]

$$\begin{aligned}\frac{\partial C}{\partial w^L} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L} \\ &= \delta_L \cdot a^{L-1} \cdot (1 - a^{L-1}) \cdot a^{L-1}\end{aligned} \tag{A.3}$$

In Equation A.3, the term $\frac{\partial C}{\partial w^L}$ represents the relative amount by which the weights at layer L affect the total cost, and it is used to update and adjust the weights at this particular layer in the aim of reducing the cost. [15]

The *back-propagation* process is then continued down the rest of the layers, and to carry on with equations A.1, A.2, and A.3 for layer $(L-1)$ and for the sake of some continuity this will now be explored in A.4 as before for layer L. Notice how the total error δ_L is being back-propagated to layer $(L-1)$. [15]

$$\begin{aligned}\delta_{L-1} &= \frac{\partial C}{\partial a^{L-1}} \\ &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \\ &= \delta_L \cdot a^L(1 - a^L) \cdot w^L\end{aligned} \tag{A.4}$$

Then, again but for layer $(L-1)$, in order to adjust the weights in layer $(L-1)$ we find the corresponding term $\frac{\partial C}{\partial w^{L-1}}$ with Equation A.5. Every step so far is repeated for every layer backwards until the *first hidden layer*. [15]

$$\begin{aligned}\frac{\partial C}{\partial w^{L-1}} &= \frac{\partial C}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}} \\ &= \delta_{L-1} \cdot a^{L-1} \cdot (1 - a^{L-1}) \cdot a^{L-2}\end{aligned} \tag{A.5}$$

[1] The derivative of $f(g(x))$ is equal to $f'(g(x)) \cdot g'(x)$.

So far in this section, the simplified notation has been used where we don't represent each neuron and each input and each output and each hidden layer and so on, thus, in order to have a general idea how everything discussed in this section can be generalized. This will now be demonstrated in Equation A.6 by revisiting the total cost from Equation A.1.

$$C_0 = \sum_{i=1}^{n} (a_i^L - y_i)^2 \tag{A.6}$$

Notice that in Equation A.6, n refers to the number of neurons in the given layer L. In the same Equation A.6, the a_n^L is layer L activation vector consisting of the activation of each neuron in layer L while the y_n is the output vector consisting of every neuron's output value in the output layer L.

To further improve the notation to suit the applicability of all the back-propagation equations to typical network models, it can be noticed that referring to specific individual weights requires defining the two neurons that are connected with this particular weighted connection. To solve this, it is stated that i is the index of neurons in the final hidden layer while j is the index of the output layer. This way, a matrix of weights that can be assessed with a row for each output neuron and a column for each hidden layer neuron and thus each weight can now be denoted by w_{ji}. [15]

Equation A.7 describes the gradient of the cost with regards to each weight[2] in layer L.

How does this affect Equation A.3 is expressed in Equation A.7.

$$\frac{\partial C}{\partial w_{ji}^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{ji}^L} \tag{A.7}$$

Equation A.7 can now be applied on every weight in the given layer layer L creating the gradient vector for the weights of size $j \cdot i$. Consequently, Equation A.4 that expresses the gradient of the cost with regards to the preceding layer's activation/output a_{L-1} becomes expressed as in Equation A.8

[2] There are $j \cdot i$ weights, one for each connection between layer L neurons and layer $L-1$ neurons.

$$\begin{aligned}\delta_{L-1} &= \frac{\partial C}{\partial a_i^{L-1}} \\ &= \sum_{j=0}^{n_j-1} \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial a_i^{L-1}}\end{aligned} \tag{A.8}$$

A.2 Trained models overview

Throughout the training phase of a model that actualizes this work's requirements, several models were trained. All models employed the Adam optimizer function. Some of those models varied in multiple characteristics and it will now be explored in Table A.1. Note that the accuracy here refers to the validation accuracy.

The Table A.1 does not give a complete picture of the variation between the trained models. Therefore, this section will also contain the structural summary of each trained model, along with its corresponding accuracy and loss plots.

Model	Accuracy	Learning rate	Batch size	Data augmentation	Batch normalization
1	55%	0.0001	32	No	No
2	72%	0.0005	32	No	No
7	85%	0.001	32	No	Yes
8	87%	0.001	24	Yes	Yes
9	90%	0.0001	16	Yes	Yes
10	90%	0.0001	16	Yes	Yes
12	89%	0.001	24	Yes	Yes
13	90%	0.001	24	Yes	Yes

Table A.1: Trained models

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 638, 478, 12)      336

 conv2d_1 (Conv2D)           (None, 636, 476, 14)      1526

 conv2d_2 (Conv2D)           (None, 634, 474, 16)      2032

 max_pooling2d (MaxPooling2D  (None, 317, 237, 16)     0
 )

 conv2d_3 (Conv2D)           (None, 315, 235, 18)      2610

 conv2d_4 (Conv2D)           (None, 313, 233, 20)      3260

 conv2d_5 (Conv2D)           (None, 311, 231, 22)      3982

 max_pooling2d_1 (MaxPooling  (None, 155, 115, 22)     0
 2D)

 conv2d_6 (Conv2D)           (None, 153, 113, 24)      4776

 conv2d_7 (Conv2D)           (None, 151, 111, 18)      3906

 conv2d_8 (Conv2D)           (None, 149, 109, 12)      1956

 max_pooling2d_2 (MaxPooling  (None, 74, 54, 12)       0
 2D)

 conv2d_9 (Conv2D)           (None, 72, 52, 6)         654

 max_pooling2d_3 (MaxPooling  (None, 36, 26, 6)        0
 2D)

 flatten (Flatten)           (None, 5616)              0

 dense (Dense)               (None, 20)                112340

=================================================================
Total params: 137,378
Trainable params: 137,378
Non-trainable params: 0
_________________________________________________________________
```

Figure A.1: Model 1 summary

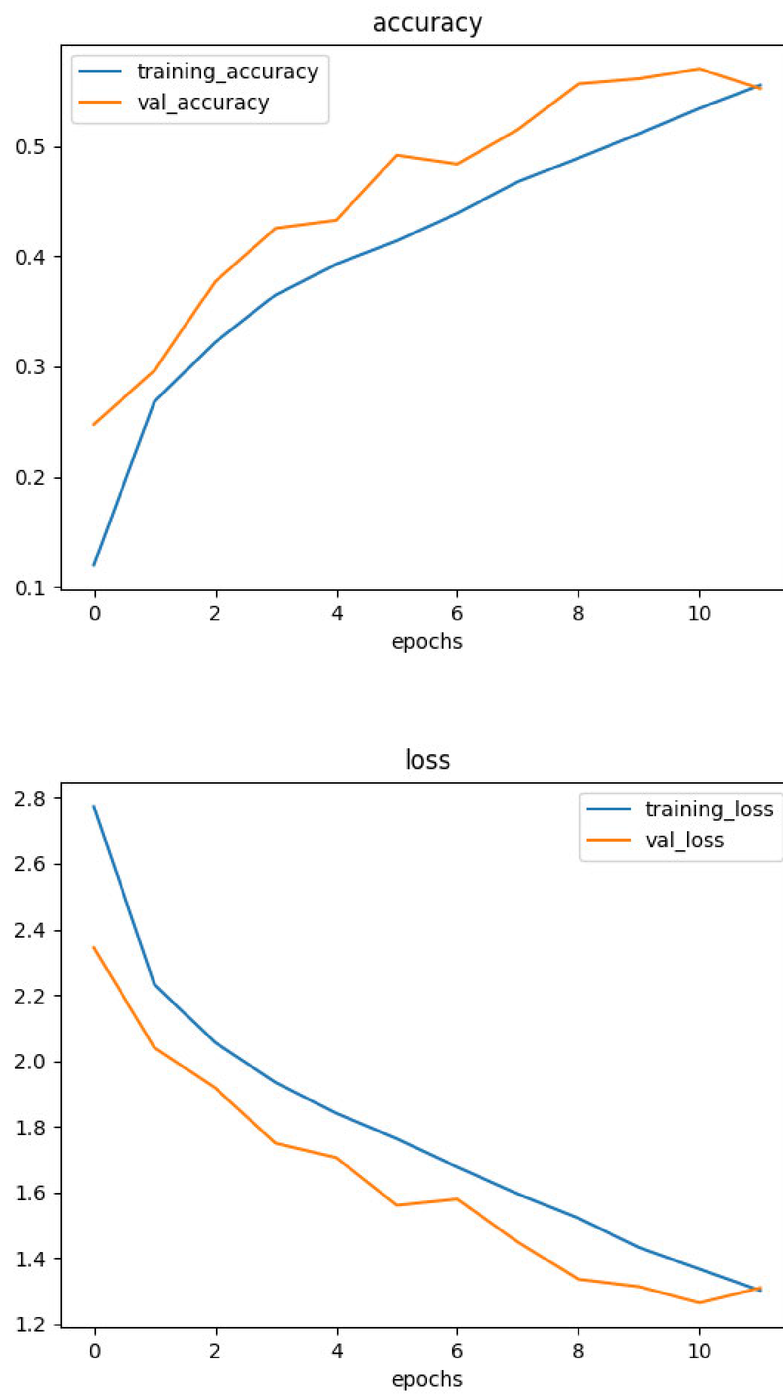


Figure A.2: Model 1 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 638, 478, 12)      336

 conv2d_1 (Conv2D)           (None, 636, 476, 14)      1526

 conv2d_2 (Conv2D)           (None, 634, 474, 16)      2032

 max_pooling2d (MaxPooling2D  (None, 317, 237, 16)     0
 )

 conv2d_3 (Conv2D)           (None, 315, 235, 22)      3190

 max_pooling2d_1 (MaxPooling  (None, 157, 117, 22)     0
 2D)

 conv2d_4 (Conv2D)           (None, 155, 115, 16)      3184

 max_pooling2d_2 (MaxPooling  (None, 77, 57, 16)       0
 2D)

 conv2d_5 (Conv2D)           (None, 75, 55, 8)         1160

 max_pooling2d_3 (MaxPooling  (None, 37, 27, 8)        0
 2D)

 flatten (Flatten)           (None, 7992)              0

 dense (Dense)               (None, 20)                159860

=================================================================
Total params: 171,288
Trainable params: 171,288
Non-trainable params: 0
_________________________________________________________________
None
```

Figure A.3: Model 2 summary

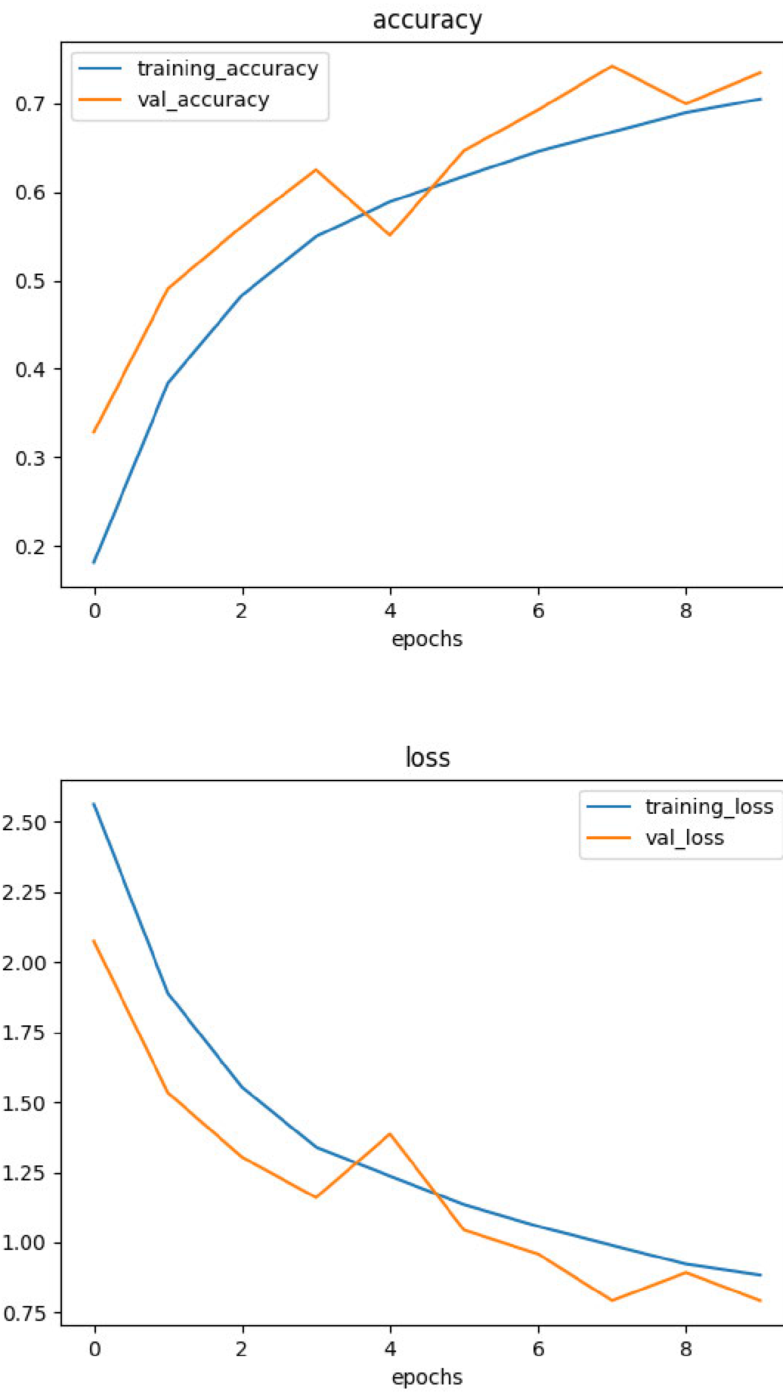


Figure A.4: Model 2 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 640, 480, 8)       224

 conv2d_1 (Conv2D)           (None, 638, 478, 8)       584

 conv2d_2 (Conv2D)           (None, 636, 476, 8)       584

 batch_normalization (BatchN  (None, 636, 476, 8)      32
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 159, 119, 8)      0
 )

 conv2d_3 (Conv2D)           (None, 157, 117, 10)      730

 conv2d_4 (Conv2D)           (None, 155, 115, 10)      910

 batch_normalization_1 (Batc  (None, 155, 115, 10)     40
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 77, 57, 10)       0
 2D)

 conv2d_5 (Conv2D)           (None, 75, 55, 12)        1092

 conv2d_6 (Conv2D)           (None, 73, 53, 12)        1308

 batch_normalization_2 (Batc  (None, 73, 53, 12)       48
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 36, 26, 12)       0
 2D)

 conv2d_7 (Conv2D)           (None, 34, 24, 14)        1526

 conv2d_8 (Conv2D)           (None, 32, 22, 14)        1778

 batch_normalization_3 (Batc  (None, 32, 22, 14)       56
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 16, 11, 14)       0
 2D)

 conv2d_9 (Conv2D)           (None, 14, 9, 16)         2032

 conv2d_10 (Conv2D)          (None, 12, 7, 16)         2320

 conv2d_11 (Conv2D)          (None, 10, 5, 16)         2320

 batch_normalization_4 (Batc  (None, 10, 5, 16)        64
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 5, 2, 16)         0
 2D)

 flatten (Flatten)           (None, 160)               0

 dense (Dense)               (None, 120)               19320

 batch_normalization_5 (Batc  (None, 120)              480
 hNormalization)

 dense_1 (Dense)             (None, 90)                10890

 batch_normalization_6 (Batc  (None, 90)               360
 hNormalization)

 dense_2 (Dense)             (None, 60)                5460

 dense_3 (Dense)             (None, 20)                1220

=================================================================
Total params: 53,378
Trainable params: 52,838
Non-trainable params: 540
_________________________________________________________________
None
Epoch 1/20
```

Figure A.5: Model 3 summary

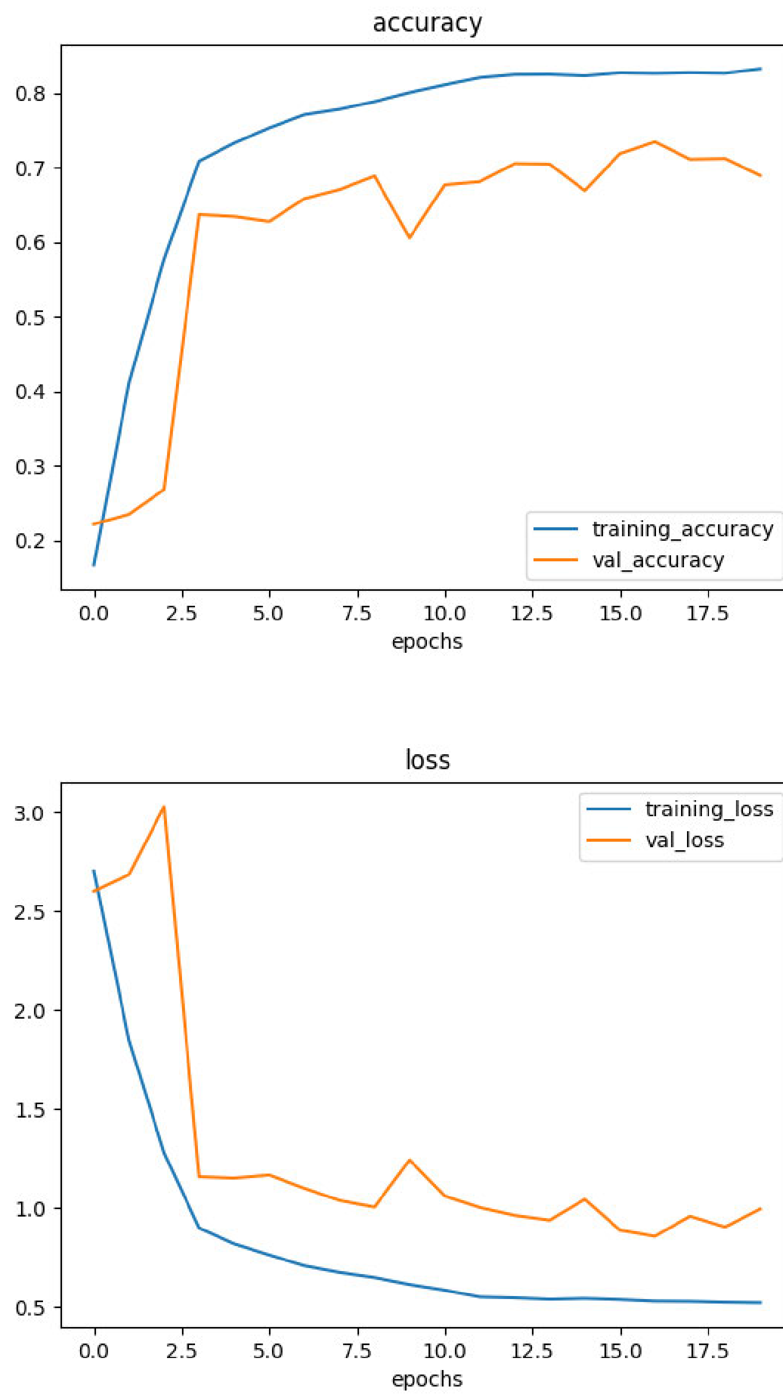


Figure A.6: Model 3 accuracy and loss plots

```
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 638, 478, 24)      672

 conv2d_1 (Conv2D)           (None, 636, 476, 24)      5208

 max_pooling2d (MaxPooling2D  (None, 159, 119, 24)     0
 )

 conv2d_2 (Conv2D)           (None, 157, 117, 24)      5208

 conv2d_3 (Conv2D)           (None, 155, 115, 24)      5208

 max_pooling2d_1 (MaxPooling  (None, 51, 38, 24)       0
 2D)

 conv2d_4 (Conv2D)           (None, 49, 36, 24)        5208

 conv2d_5 (Conv2D)           (None, 47, 34, 24)        5208

 conv2d_6 (Conv2D)           (None, 45, 32, 24)        5208

 max_pooling2d_2 (MaxPooling  (None, 22, 16, 24)       0
 2D)

 flatten (Flatten)           (None, 8448)              0

 dense (Dense)               (None, 64)                540736

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 20)                660

=================================================================
Total params: 579,556
Trainable params: 579,556
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/20
2022-06-13 21:21:18.541501: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8100
2022-06-13 21:21:19.259648: W tensorflow/core/common_runtime/bfc_allocator.cc:290] Allocator (GPU_0_bfc) ran out of memory trying to allocate 3.91GiB with free
an that there could be performance gains if more memory were available.
2121/2121 [==============================] - ETA: 0s - loss: 2.6745 - accuracy: 0.1328WARNING:absl:Found untraced functions such as _jit_compiled_convolution_o
onvolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
2121/2121 [==============================] - 1348s 634ms/step - loss: 2.6745 - accuracy: 0.1328 - val_loss: 2.3421 - val_accuracy: 0.1959 - lr: 1.0000e-04
Epoch 2/20
```

Figure A.7: Model 4 summary

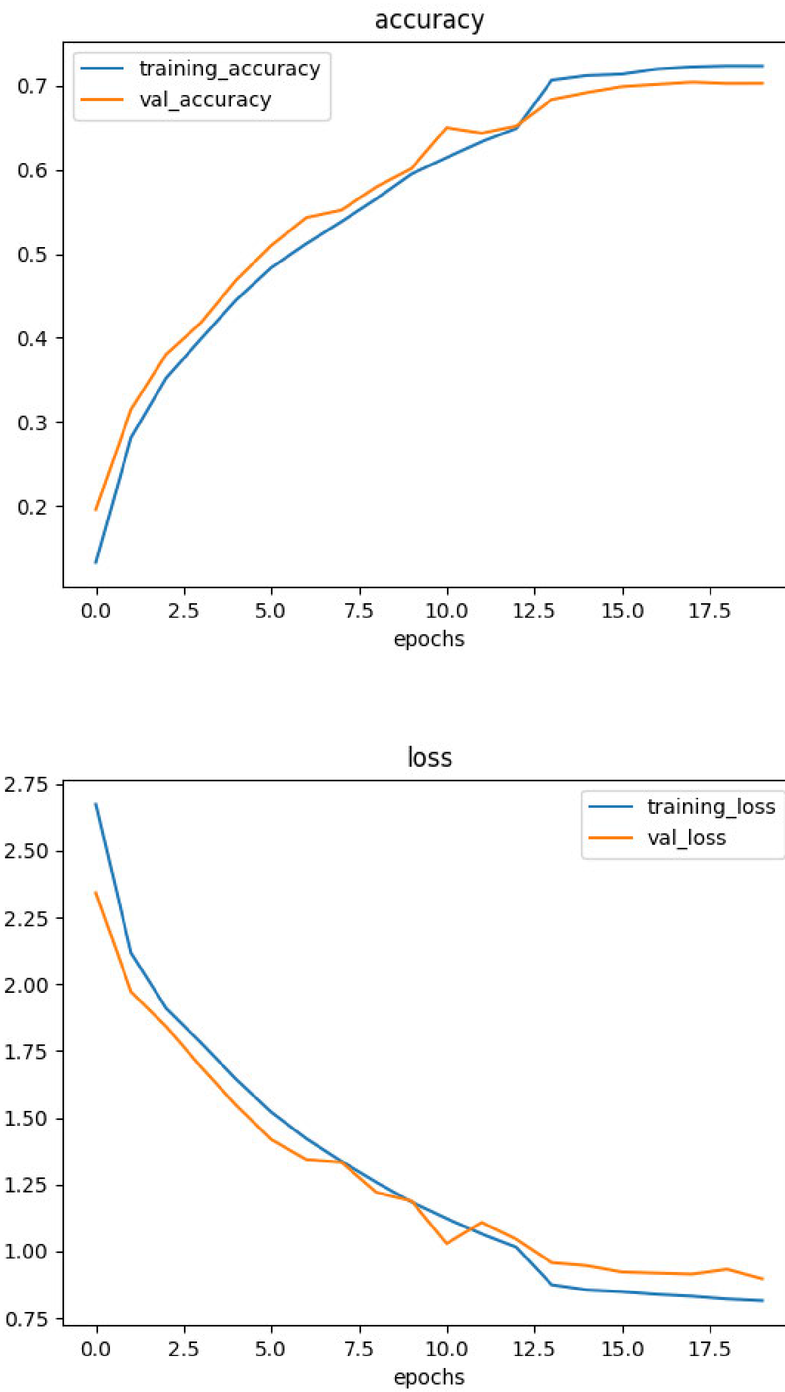


Figure A.8: Model 4 accuracy and loss plots

```
random_flip (RandomFlip)     (None, 640, 480, 3)       0

random_rotation (RandomRota  (None, 640, 480, 3)       0
tion)

random_zoom (RandomZoom)     (None, 640, 480, 3)       0

conv2d (Conv2D)              (None, 638, 478, 14)      392

batch_normalization (BatchN  (None, 638, 478, 14)      56
ormalization)

conv2d_1 (Conv2D)            (None, 636, 476, 14)      1778

batch_normalization_1 (Batc  (None, 636, 476, 14)      56
hNormalization)

max_pooling2d (MaxPooling2D  (None, 106, 79, 14)       0
)

conv2d_2 (Conv2D)            (None, 104, 77, 16)       2032

batch_normalization_2 (Batc  (None, 104, 77, 16)       64
hNormalization)

conv2d_3 (Conv2D)            (None, 102, 75, 16)       2320

batch_normalization_3 (Batc  (None, 102, 75, 16)       64
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 34, 25, 16)        0
2D)

conv2d_4 (Conv2D)            (None, 32, 23, 18)        2610

batch_normalization_4 (Batc  (None, 32, 23, 18)        72
hNormalization)

conv2d_5 (Conv2D)            (None, 30, 21, 18)        2934

batch_normalization_5 (Batc  (None, 30, 21, 18)        72
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 15, 10, 18)        0
2D)

flatten (Flatten)            (None, 2700)              0

dense (Dense)                (None, 128)               345728

batch_normalization_6 (Batc  (None, 128)               512
hNormalization)

dense_1 (Dense)              (None, 96)                12384

batch_normalization_7 (Batc  (None, 96)                384
hNormalization)

dense_2 (Dense)              (None, 64)                6208

batch_normalization_8 (Batc  (None, 64)                256
hNormalization)

dense_3 (Dense)              (None, 32)                2080

batch_normalization_9 (Batc  (None, 32)                128
hNormalization)

dense_4 (Dense)              (None, 20)                660

=================================================================
Total params: 380,790
Trainable params: 379,958
Non-trainable params: 832
_________________________________________________________________
None
Epoch 1/16
```

Figure A.9: Model 5 summary

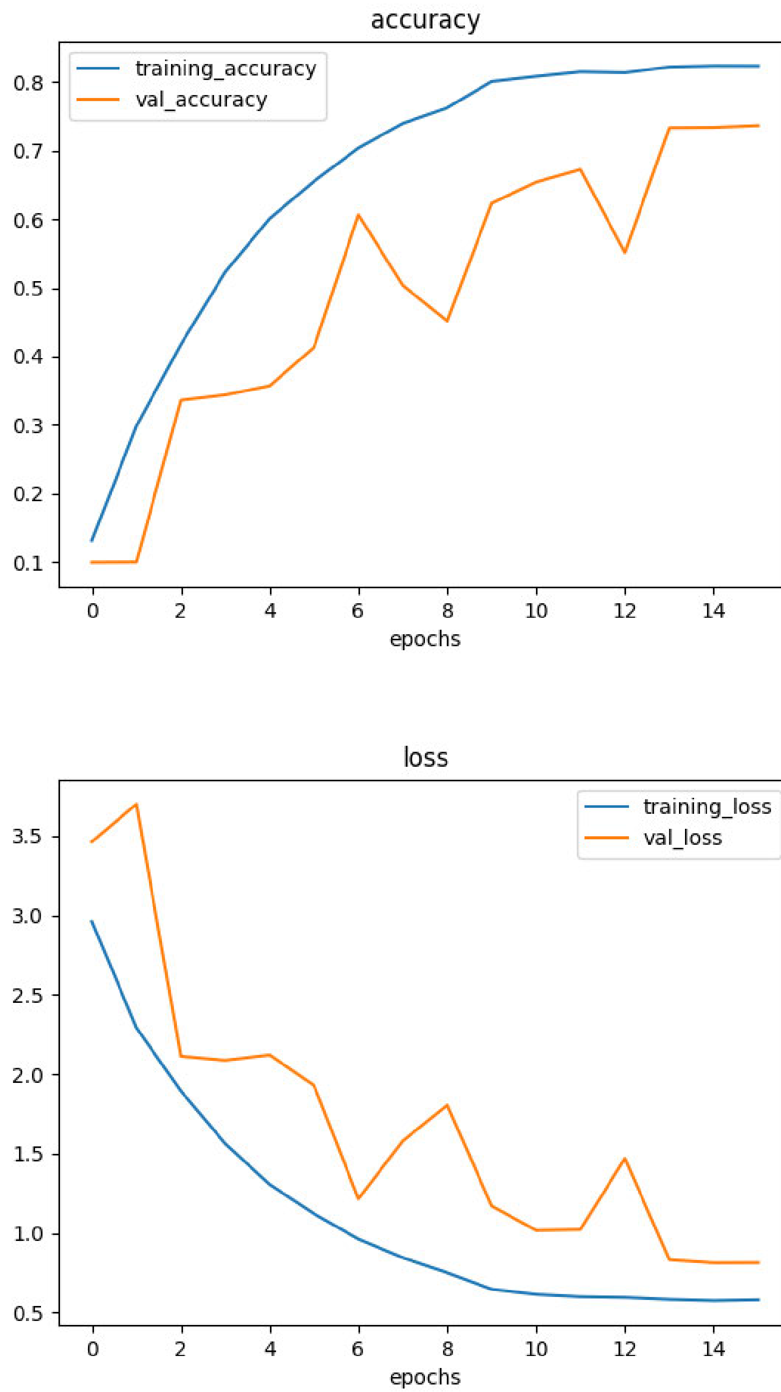


Figure A.10: Model 5 accuracy and loss plots

```
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 638, 478, 24)      672

 conv2d_1 (Conv2D)           (None, 636, 476, 24)      5208

 max_pooling2d (MaxPooling2D  (None, 159, 119, 24)     0
 )

 conv2d_2 (Conv2D)           (None, 157, 117, 24)      5208

 conv2d_3 (Conv2D)           (None, 155, 115, 24)      5208

 conv2d_4 (Conv2D)           (None, 153, 113, 24)      5208

 max_pooling2d_1 (MaxPooling  (None, 51, 37, 24)       0
 2D)

 conv2d_5 (Conv2D)           (None, 49, 35, 24)        5208

 conv2d_6 (Conv2D)           (None, 47, 33, 24)        5208

 conv2d_7 (Conv2D)           (None, 45, 31, 24)        5208

 max_pooling2d_2 (MaxPooling  (None, 22, 15, 24)       0
 2D)

 flatten (Flatten)           (None, 7920)              0

 dense (Dense)               (None, 96)                760416

 dense_1 (Dense)             (None, 64)                6208

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 20)                660

=================================================================
Total params: 806,492
Trainable params: 806,492
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/36
2022-06-15 10:06:55.574523: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuD
2022-06-15 10:06:59.006633: W tensorflow/core/common_runtime/bfc_allocator.cc:290] Alloca
an that there could be performance gains if more memory were available.
  40/2121 [..............................] - ETA: 26:37 - loss: 2.9975 - accuracy: 0.0500
```

Figure A.11: Model 6 summary

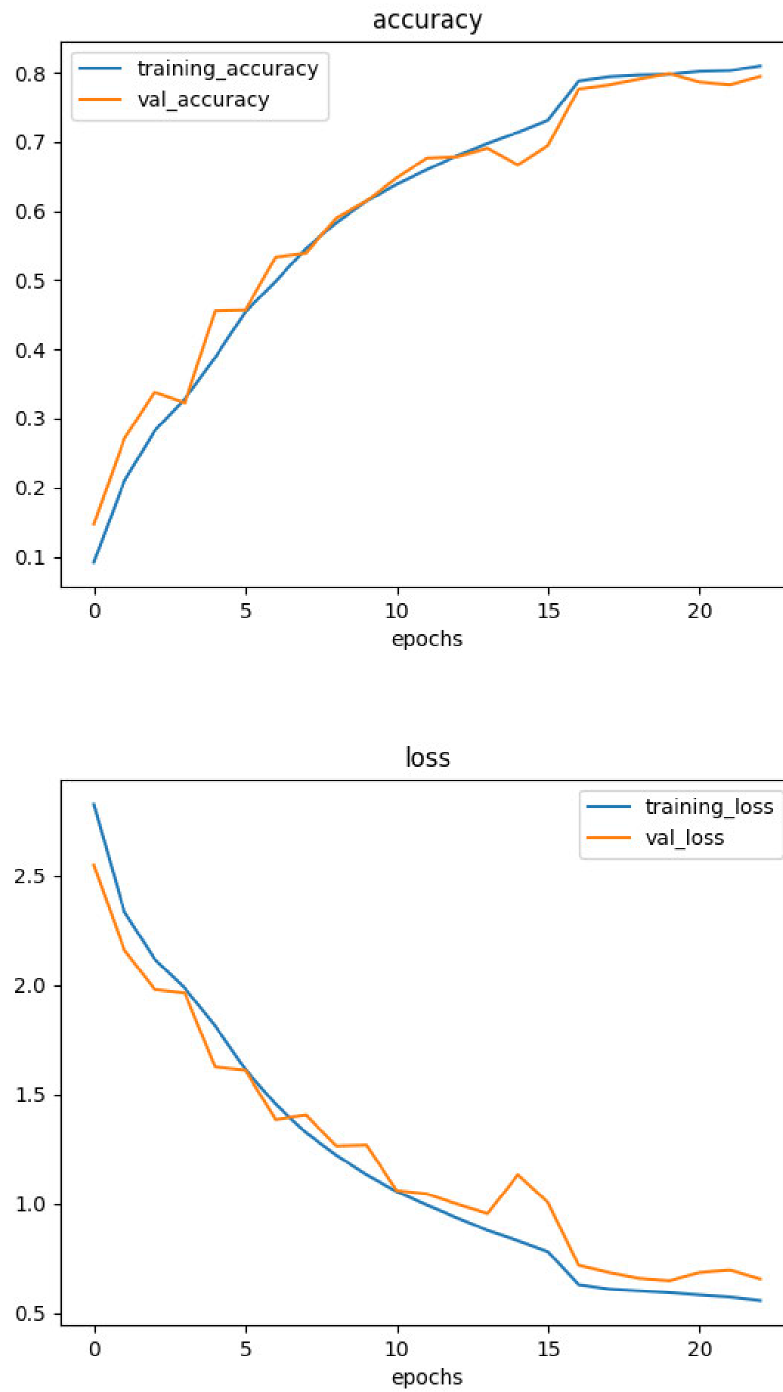


Figure A.12: Model 6 accuracy and loss plots

```
Layer (type)                Output Shape              Param #
=================================================================
random_flip (RandomFlip)    (None, 640, 480, 3)       0

random_rotation (RandomRota  (None, 640, 480, 3)      0
tion)

random_zoom (RandomZoom)    (None, 640, 480, 3)       0

conv2d (Conv2D)             (None, 320, 240, 20)      1520

conv2d_1 (Conv2D)           (None, 318, 238, 20)      3620

batch_normalization (BatchN  (None, 318, 238, 20)     80
ormalization)

max_pooling2d (MaxPooling2D  (None, 79, 59, 20)       0
)

conv2d_2 (Conv2D)           (None, 77, 57, 40)        7240

conv2d_3 (Conv2D)           (None, 75, 55, 40)        14440

batch_normalization_1 (Batc  (None, 75, 55, 40)       160
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 25, 18, 40)       0
2D)

conv2d_4 (Conv2D)           (None, 23, 16, 60)        21660

conv2d_5 (Conv2D)           (None, 21, 14, 60)        32460

conv2d_6 (Conv2D)           (None, 19, 12, 60)        32460

batch_normalization_2 (Batc  (None, 19, 12, 60)       240
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 9, 6, 60)         0
2D)

flatten (Flatten)           (None, 3240)              0

dense (Dense)               (None, 120)               388920

batch_normalization_3 (Batc  (None, 120)              480
hNormalization)

dense_1 (Dense)             (None, 90)                10890

dense_2 (Dense)             (None, 90)                8190

dense_3 (Dense)             (None, 60)                5460

batch_normalization_4 (Batc  (None, 60)               240
hNormalization)

dense_4 (Dense)             (None, 20)                1220

=================================================================
Total params: 529,280
Trainable params: 528,680
Non-trainable params: 600
_________________________________________________________________
None
Epoch 1/20
```

Figure A.13: Model 7 summary

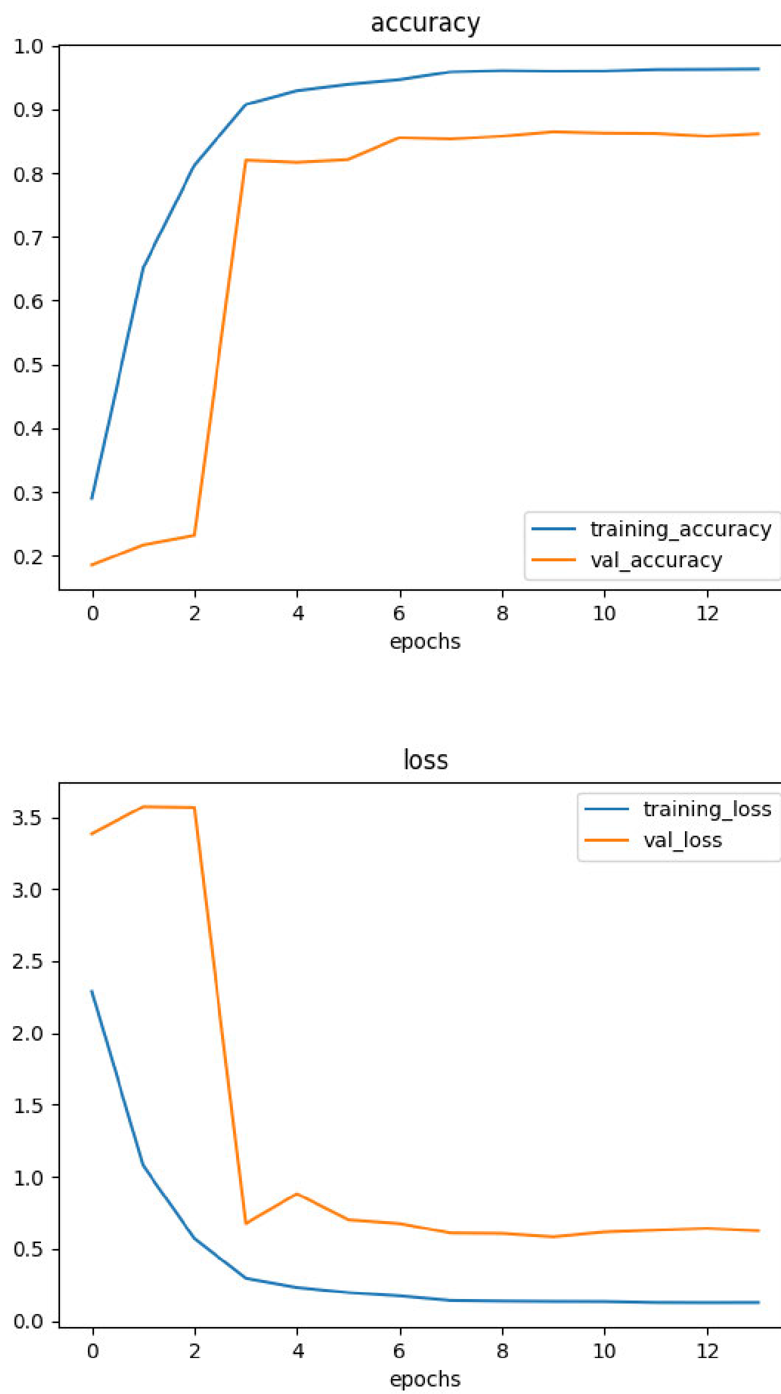


Figure A.14: Model 7 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 640, 480, 16)      448

 conv2d_1 (Conv2D)           (None, 638, 478, 16)      2320

 conv2d_2 (Conv2D)           (None, 636, 476, 16)      2320

 batch_normalization (BatchN  (None, 636, 476, 16)     64
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 159, 119, 16)     0
 )

 conv2d_3 (Conv2D)           (None, 157, 117, 18)      2610

 conv2d_4 (Conv2D)           (None, 155, 115, 18)      2934

 batch_normalization_1 (Batc  (None, 155, 115, 18)     72
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 38, 28, 18)       0
 2D)

 conv2d_5 (Conv2D)           (None, 36, 26, 20)        3260

 conv2d_6 (Conv2D)           (None, 34, 24, 20)        3620

 conv2d_7 (Conv2D)           (None, 32, 22, 20)        3620

 batch_normalization_2 (Batc  (None, 32, 22, 20)       80
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 16, 11, 20)       0
 2D)

 flatten (Flatten)           (None, 3520)              0

 dense (Dense)               (None, 120)               422520

 batch_normalization_3 (Batc  (None, 120)              480
 hNormalization)

 dense_1 (Dense)             (None, 80)                9680

 batch_normalization_4 (Batc  (None, 80)               320
 hNormalization)

 dense_2 (Dense)             (None, 80)                6480

 batch_normalization_5 (Batc  (None, 80)               320
 hNormalization)

 dense_3 (Dense)             (None, 40)                3240

 batch_normalization_6 (Batc  (None, 40)               160
 hNormalization)

 dense_4 (Dense)             (None, 20)                820

=================================================================
Total params: 465,368
Trainable params: 464,620
Non-trainable params: 748
_________________________________________________________________
None
Epoch 1/20
```

Figure A.15: Model 8 summary

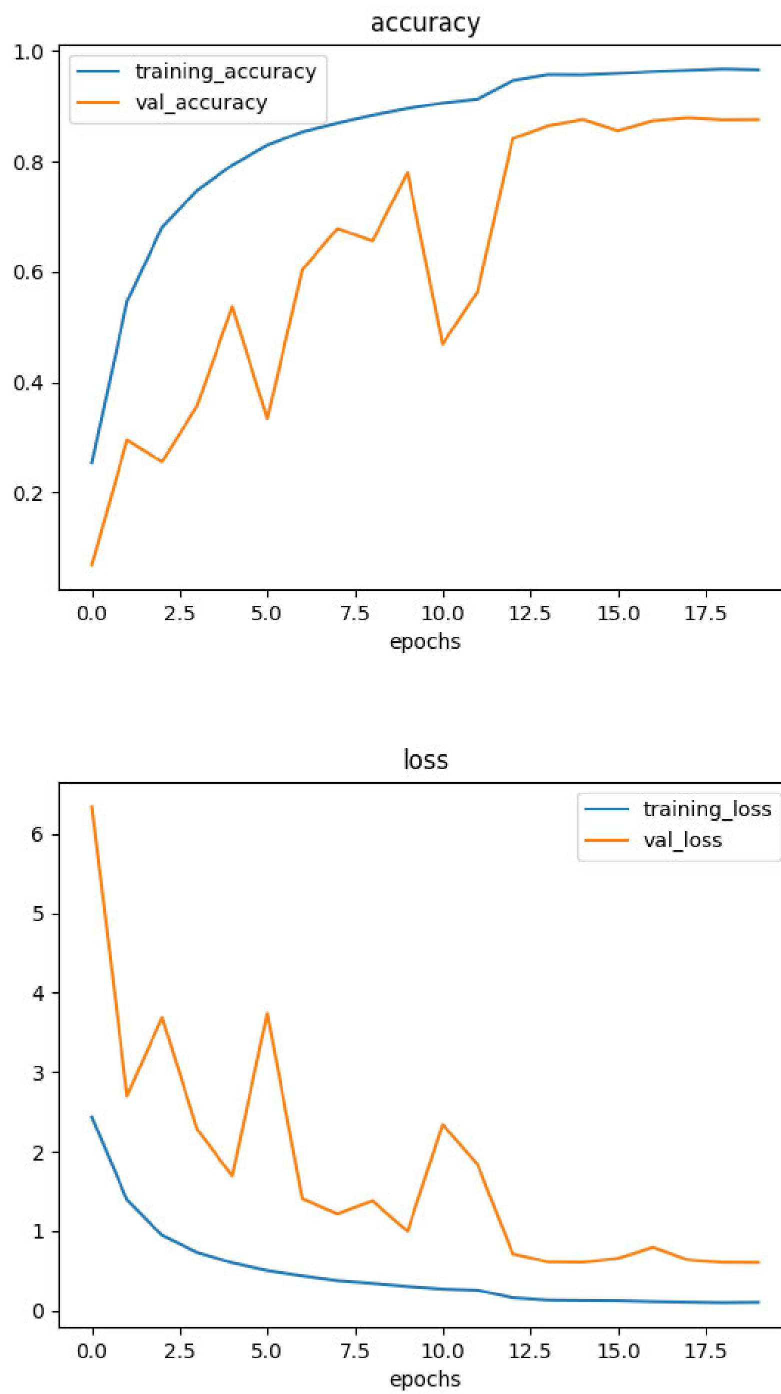


Figure A.16: Model 8 accuracy and loss plots

```
['1' '10' '100' '1000' '10000' '100000' '1000000' '150' '1500' '220'
 '2200' '22000' '220000' '330' '33000' '47' '470' '4700' '47000' '470000']
--------------------     1
Found 33928 images belonging to 20 classes.
Found 8441 images belonging to 20 classes.
2022-06-15 20:48:31.400949: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operation
s:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-06-15 20:48:32.241498: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1532] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 4632 MB memory:  -> device: 0, name: NVIDIA GeForce GTX 1060, pci bus id: 0000:01:00.0,
 compute capability: 6.1
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 638, 478, 24)      672

 batch_normalization (BatchN  (None, 638, 478, 24)     96
 ormalization)

 conv2d_1 (Conv2D)           (None, 636, 476, 24)      5208

 batch_normalization_1 (Batc  (None, 636, 476, 24)     96
 hNormalization)

 max_pooling2d (MaxPooling2D  (None, 127, 95, 24)      0
 )

 conv2d_2 (Conv2D)           (None, 125, 93, 28)       6076

 batch_normalization_2 (Batc  (None, 125, 93, 28)      112
 hNormalization)

 conv2d_3 (Conv2D)           (None, 123, 91, 28)       7084

 batch_normalization_3 (Batc  (None, 123, 91, 28)      112
 hNormalization)

 conv2d_4 (Conv2D)           (None, 121, 89, 28)       7084

 batch_normalization_4 (Batc  (None, 121, 89, 28)      112
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 40, 29, 28)       0
 2D)

 conv2d_5 (Conv2D)           (None, 38, 27, 28)        7084

 batch_normalization_5 (Batc  (None, 38, 27, 28)       112
 hNormalization)

 conv2d_6 (Conv2D)           (None, 36, 25, 28)        7084

 batch_normalization_6 (Batc  (None, 36, 25, 28)       112
 hNormalization)

 conv2d_7 (Conv2D)           (None, 34, 23, 28)        7084

 batch_normalization_7 (Batc  (None, 34, 23, 28)       112
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 17, 11, 28)       0
 2D)

 flatten (Flatten)           (None, 5236)              0

 dense (Dense)               (None, 96)                502752

 batch_normalization_8 (Batc  (None, 96)               384
 hNormalization)

 dense_1 (Dense)             (None, 64)                6208

 dense_2 (Dense)             (None, 64)                4160

 batch_normalization_9 (Batc  (None, 64)               256
 hNormalization)

 dense_3 (Dense)             (None, 32)                2080

 batch_normalization_10 (Bat  (None, 32)               128
 chNormalization)

 dense_4 (Dense)             (None, 20)                660

=================================================================
Total params: 564,868
Trainable params: 564,052
Non-trainable params: 816
_________________________________________________________________
None
Epoch 1/28
```

Figure A.17: Model 9 summary

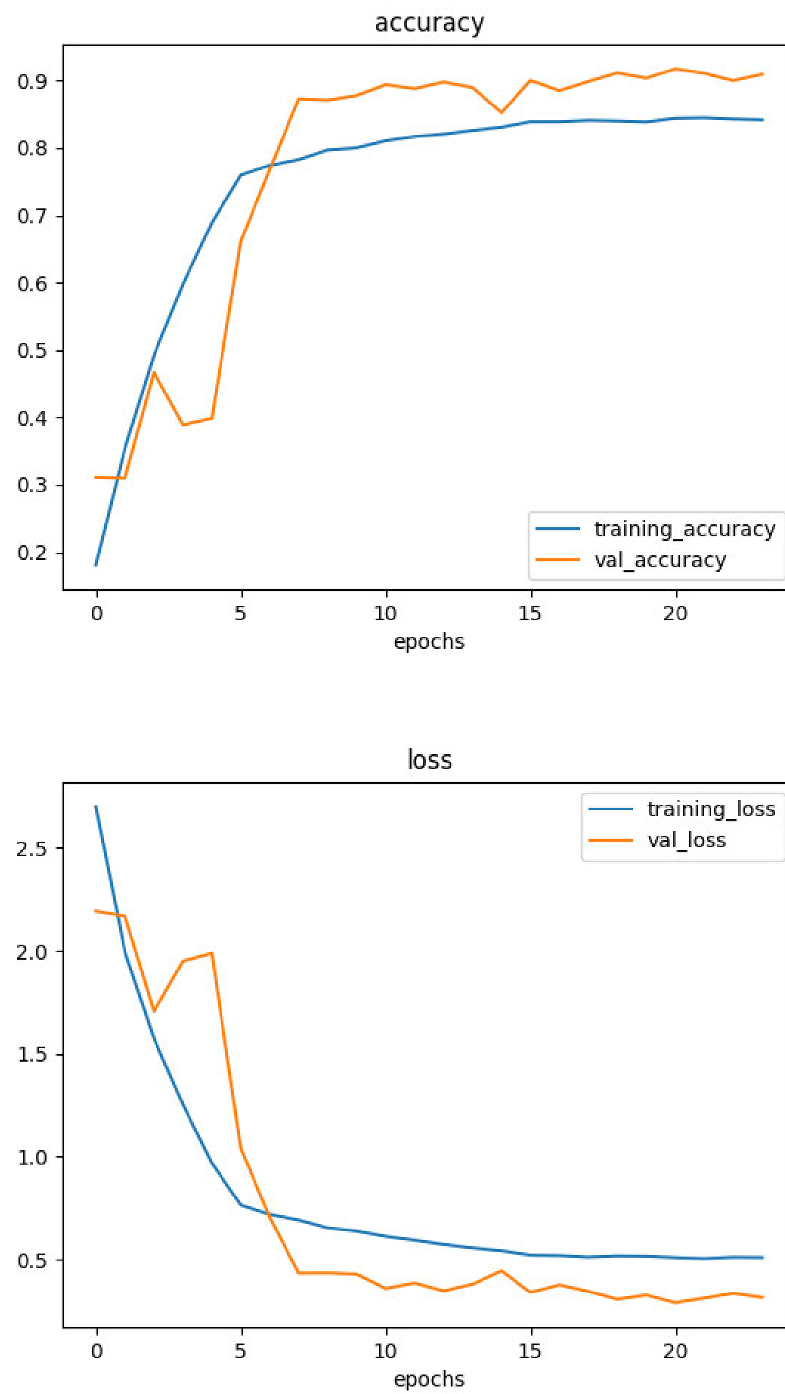


Figure A.18: Model 9 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 638, 478, 24)      672

 batch_normalization (BatchN  (None, 638, 478, 24)     96
 ormalization)

 conv2d_1 (Conv2D)           (None, 636, 476, 24)      5208

 batch_normalization_1 (Batc  (None, 636, 476, 24)     96
 hNormalization)

 conv2d_2 (Conv2D)           (None, 634, 474, 24)      5208

 batch_normalization_2 (Batc  (None, 634, 474, 24)     96
 hNormalization)

 max_pooling2d (MaxPooling2D  (None, 105, 79, 24)      0
 )

 conv2d_3 (Conv2D)           (None, 103, 77, 28)       6076

 batch_normalization_3 (Batc  (None, 103, 77, 28)      112
 hNormalization)

 conv2d_4 (Conv2D)           (None, 101, 75, 28)       7084

 batch_normalization_4 (Batc  (None, 101, 75, 28)      112
 hNormalization)

 conv2d_5 (Conv2D)           (None, 99, 73, 28)        7084

 batch_normalization_5 (Batc  (None, 99, 73, 28)       112
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 33, 24, 28)       0
 2D)

 conv2d_6 (Conv2D)           (None, 31, 22, 28)        7084

 batch_normalization_6 (Batc  (None, 31, 22, 28)       112
 hNormalization)

 conv2d_7 (Conv2D)           (None, 29, 20, 28)        7084

 batch_normalization_7 (Batc  (None, 29, 20, 28)       112
 hNormalization)

 conv2d_8 (Conv2D)           (None, 27, 18, 28)        7084

 batch_normalization_8 (Batc  (None, 27, 18, 28)       112
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 13, 9, 28)        0
 2D)

 flatten (Flatten)           (None, 3276)              0

 dense (Dense)               (None, 96)                314592

 batch_normalization_9 (Batc  (None, 96)               384
 hNormalization)

 dense_1 (Dense)             (None, 64)                6208

 dropout (Dropout)           (None, 64)                0

 dense_2 (Dense)             (None, 64)                4160

 batch_normalization_10 (Bat  (None, 64)               256
 chNormalization)

 dense_3 (Dense)             (None, 32)                2080

 batch_normalization_11 (Bat  (None, 32)               128
 chNormalization)

 dense_4 (Dense)             (None, 20)                660

=================================================================
Total params: 382,012
Trainable params: 381,148
Non-trainable params: 864
_________________________________________________________________
None
Epoch 1/24
2022-06-16 09:22:15.984909: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 81
2022-06-16 09:22:17.296996: W tensorflow/core/common_runtime/bfc_allocator.cc:290] Allocator (GPU_0_bf
an that there could be performance gains if more memory were available.
2022-06-16 09:22:17.507664: W tensorflow/core/common_runtime/bfc_allocator.cc:290] Allocator (GPU_0_bf
an that there could be performance gains if more memory were available.
```

Figure A.19: Model 10 summary

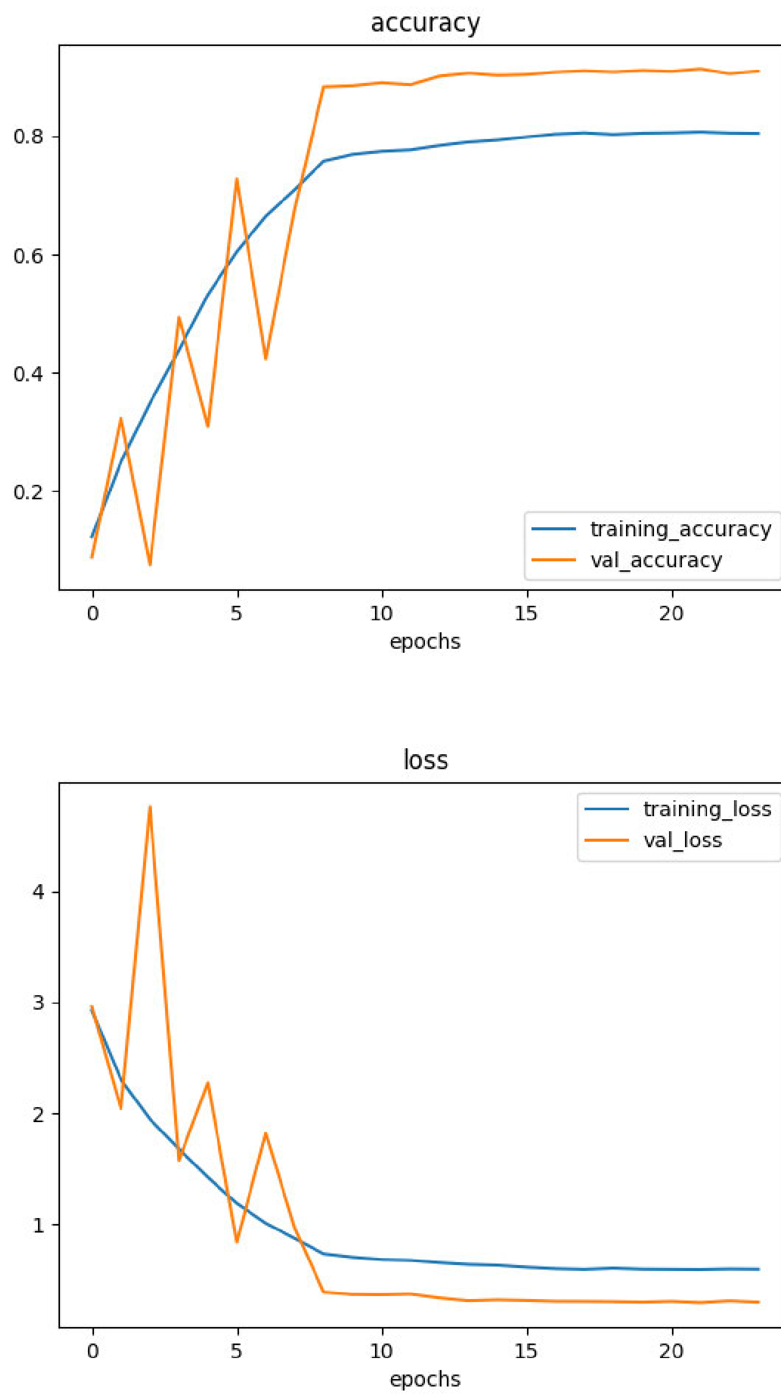


Figure A.20: Model 10 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 640, 480, 32)      896

 max_pooling2d (MaxPooling2D  (None, 213, 160, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 211, 158, 32)      9248

 max_pooling2d_1 (MaxPooling  (None, 105, 79, 32)      0
 2D)

 conv2d_2 (Conv2D)           (None, 103, 77, 48)       13872

 conv2d_3 (Conv2D)           (None, 101, 75, 48)       20784

 max_pooling2d_2 (MaxPooling  (None, 50, 37, 48)       0
 2D)

 conv2d_4 (Conv2D)           (None, 48, 35, 64)        27712

 conv2d_5 (Conv2D)           (None, 46, 33, 64)        36928

 max_pooling2d_3 (MaxPooling  (None, 23, 16, 64)       0
 2D)

 flatten (Flatten)           (None, 23552)             0

 dense (Dense)               (None, 120)               2826360

 dense_1 (Dense)             (None, 80)                9680

 dropout (Dropout)           (None, 80)                0

 dense_2 (Dense)             (None, 20)                1620

=================================================================
Total params: 2,947,100
Trainable params: 2,947,100
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/24
```

Figure A.21: Model 11 summary

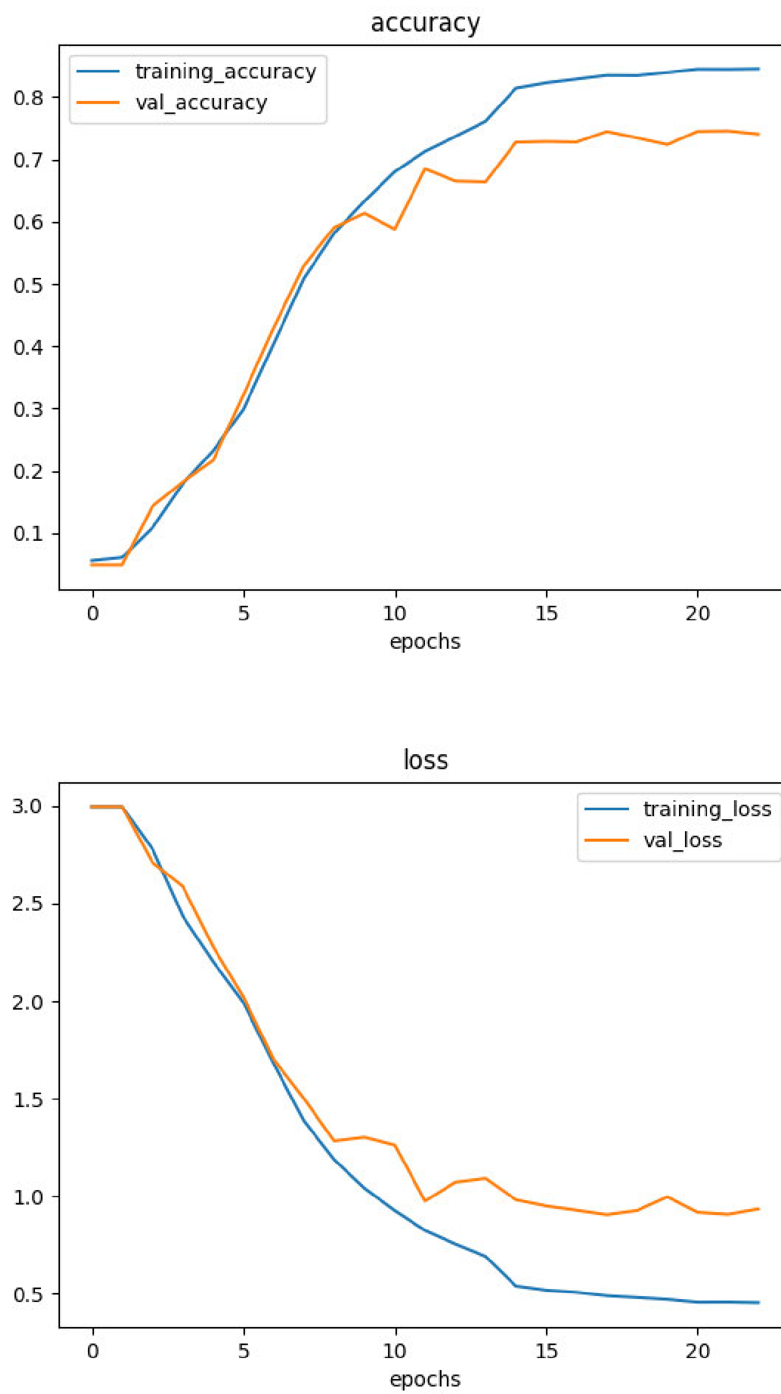


Figure A.22: Model 11 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 640, 480, 16)      448

 conv2d_1 (Conv2D)           (None, 638, 478, 16)      2320

 batch_normalization (BatchN  (None, 638, 478, 16)     64
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 319, 239, 16)     0
 )

 conv2d_2 (Conv2D)           (None, 317, 237, 32)      4640

 batch_normalization_1 (Batc  (None, 317, 237, 32)     128
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 158, 118, 32)     0
 2D)

 conv2d_3 (Conv2D)           (None, 156, 116, 32)      9248

 batch_normalization_2 (Batc  (None, 156, 116, 32)     128
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 78, 58, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 76, 56, 48)        13872

 batch_normalization_3 (Batc  (None, 76, 56, 48)       192
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 38, 28, 48)       0
 2D)

 conv2d_5 (Conv2D)           (None, 36, 26, 48)        20784

 conv2d_6 (Conv2D)           (None, 34, 24, 48)        20784

 batch_normalization_4 (Batc  (None, 34, 24, 48)       192
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 17, 12, 48)       0
 2D)

 flatten (Flatten)           (None, 9792)              0

 dense (Dense)               (None, 120)               1175160

 batch_normalization_5 (Batc  (None, 120)              480
 hNormalization)

 dense_1 (Dense)             (None, 90)                10890

 dense_2 (Dense)             (None, 20)                1820

=================================================================
Total params: 1,261,150
Trainable params: 1,260,558
Non-trainable params: 592
_________________________________________________________________
None
Epoch 1/20
```

Figure A.23: Model 12 summary

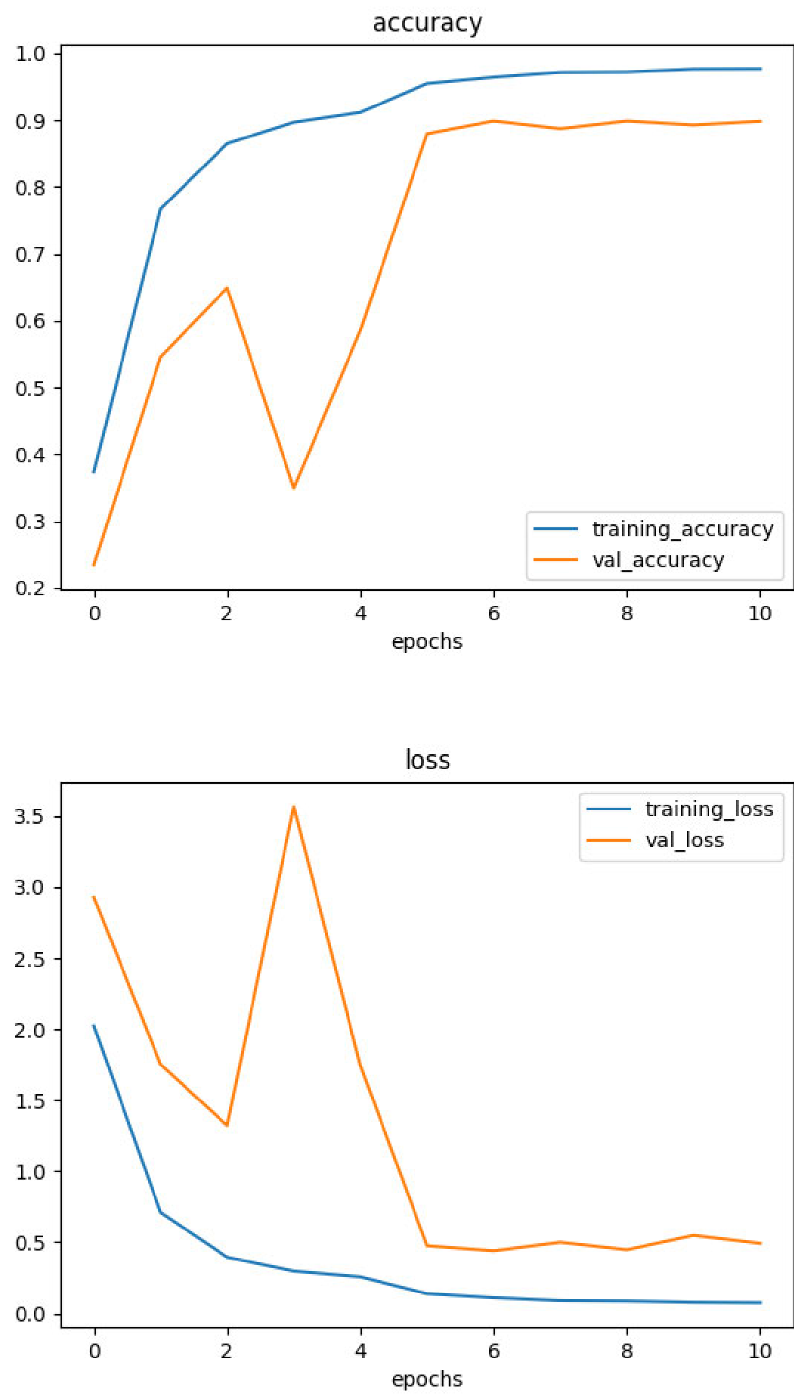


Figure A.24: Model 12 accuracy and loss plots

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 random_flip (RandomFlip)    (None, 640, 480, 3)       0

 random_rotation (RandomRota  (None, 640, 480, 3)      0
 tion)

 random_zoom (RandomZoom)    (None, 640, 480, 3)       0

 conv2d (Conv2D)             (None, 640, 480, 16)      448

 conv2d_1 (Conv2D)           (None, 638, 478, 16)      2320

 batch_normalization (BatchN  (None, 638, 478, 16)     64
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 319, 239, 16)     0
 )

 conv2d_2 (Conv2D)           (None, 317, 237, 32)      4640

 batch_normalization_1 (Batc  (None, 317, 237, 32)     128
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 158, 118, 32)     0
 2D)

 conv2d_3 (Conv2D)           (None, 156, 116, 32)      9248

 batch_normalization_2 (Batc  (None, 156, 116, 32)     128
 hNormalization)

 max_pooling2d_2 (MaxPooling  (None, 78, 58, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 76, 56, 48)        13872

 batch_normalization_3 (Batc  (None, 76, 56, 48)       192
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 38, 28, 48)       0
 2D)

 conv2d_5 (Conv2D)           (None, 36, 26, 48)        20784

 conv2d_6 (Conv2D)           (None, 34, 24, 48)        20784

 batch_normalization_4 (Batc  (None, 34, 24, 48)       192
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 17, 12, 48)       0
 2D)

 flatten (Flatten)           (None, 9792)              0

 dropout (Dropout)           (None, 9792)              0

 dense (Dense)               (None, 120)               1175160

 batch_normalization_5 (Batc  (None, 120)              480
 hNormalization)

 dense_1 (Dense)             (None, 80)                9680

 dense_2 (Dense)             (None, 20)                1620

=================================================================
Total params: 1,259,740
Trainable params: 1,259,148
Non-trainable params: 592
_________________________________________________________________
None
Epoch 1/20
2022-06-21 14:33:22.096460: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8100
2022-06-21 14:33:23.049441: W tensorflow/core/common_runtime/bfc_allocator.cc:290] Allocator (GPU_0_bfc)
```

Figure A.25: Model 13 summary

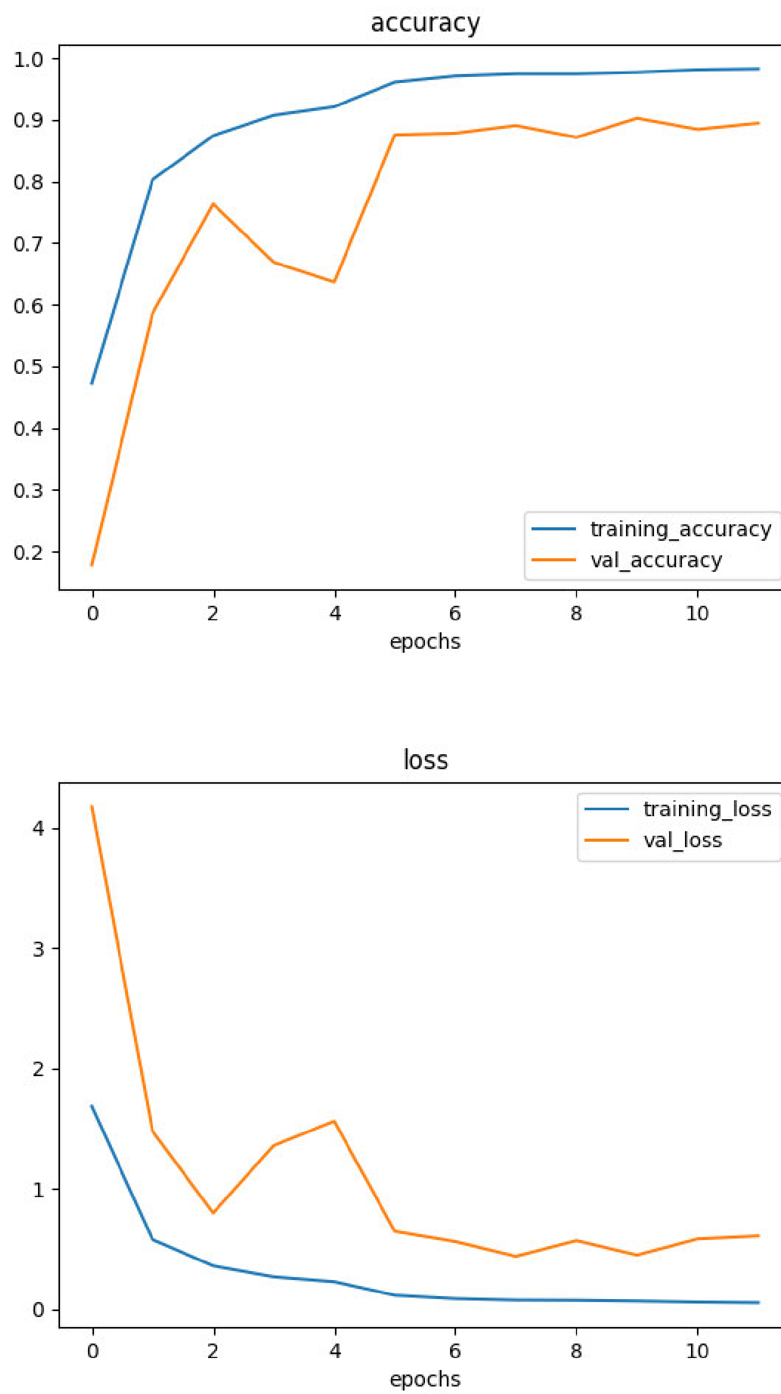
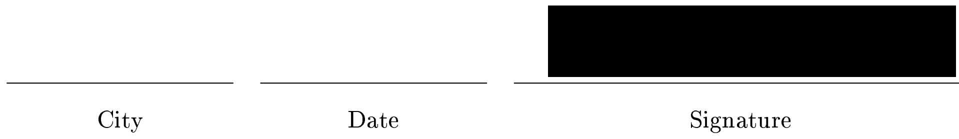


Figure A.26: Model 13 accuracy and loss plots

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.



City Date Signature